GRAPE

GRAphics Programming Environment

Manual

Sonderforschungsbereich 256
Institut für Angewandte Mathematik
Universität Bonn

Institut für Angewandte Mathematik
Universität Freiburg

Version 5.3

Contents

Preface

GRAPE is a graphics programming environment which offers a natural approach to mathematical visualization. Thereby it supports the understanding and solving of mathematical problems in different fields of research like differential geometry or continuum mechanics.

GRAPE contains a toolbox to work with geometric and numerical data, generate graphic output and film sequences. The environment is based on an object-oriented kernel and a machine independent interface for the graphics. The system gives the user interactive control in numerical algorithms combined with three dimensional computer graphics. An important aspect is time–dependency of data structures and algorithms. GRAPE is an open system: New data classes and user defined methods can easily be added.

GRAPE has been developed since 1987 at the Sonderforschungsbereich 256 for Nonlinear Partial Differential Equations at Bonn University and since 1992 in collaboration with the Institut für Angewandte Mathematik at Freiburg University. Now GRAPE is also being widely used by various research groups and in other scientific areas. As well as the number of GRAPE users has increased, GRAPE itself has been extended and improved significantly.

We hope, GRAPE will prove to be useful in some of your visualization problems.

Bonn, Freiburg · March 1997 The GRAPE–Team

E–Mail: `grape@iam.uni-bonn.de`
WWW: `http://www-sfb256.iam.uni-bonn.de/grape/main.html`
FTP: `ftp@ftp.iam.uni-bonn.de`

Chapter 1

Introduction

1.1 What is GRAPE?

GRAPE is intended to serve as a general programming and viewing environment. It benefits from its interactive approach and its standard-C based yet object-oriented structure. It has an object-oriented kernel and a library of classes and methods for different geometric objects as well as mathematical descriptions and other technical entities. Each class contains a set of methods to which standard object-oriented mechanisms like inheritance etc. apply. The class library consists of classes for triangulated geometric objects like curves, surfaces and volumes and subclasses of them with additional information like finite elements functions, refinement information, and functional descriptions. Objects can be combined like surfaces with a domain or with other maps existing on it. Nearly all static classes may be used in time-dependent classes to describe dynamic processes like deforming curves and surfaces or time-dependent vector fields on surfaces and in volumes. Non-mathematical classes exist for graphics drivers and the viewer together with its control elements such as buttons, rulers and spline editors.

The mathematical algorithms include tools to compute parametric curves and surfaces, level set surfaces, minimal and H-surfaces or the evolution of surfaces and of curves restricted to spatial surfaces, as well as tools to solve ordinary and partial differential equations, to work with vector fields on surfaces and in volumes, geometric objects in Euclidean, spherical and hyperbolic space or to evaluate finite element functions on geometric objects.

The viewer of GRAPE consists of a graphics window for graphics output and a control panel with different layers. Standard layers for lighting, surface properties, transformation operations and the data hierarchy manager are fixed. Six optional layers may be designed by the user or by invoked interactive methods with control elements like buttons, rulers, spline editors, sliders and other objects.

In addition to standard viewing facilities GRAPE offers various clipping algorithms, extraction of level lines and level surfaces, interactive refine and coarsen operations, reflection tools, as well as to calculate particle traces, to display functions on surfaces or volumes or to edit grids. Most routines work cooperatively with time-dependent geometries.

Time-dependency is a particular feature of GRAPE, introduced by extending the former static system in a natural and consistent way. Nearly all classes describing geometric objects may be time-dependent, including for example adaptive triangulations in time-direction. Also many algorithms are time-dependent, for example reflection of a time-dependent surface at a time-dependent symmetry plane.

A project management is used to organize work and enhance modularity. For example, a user who has written a small surface builder package for GRAPE consisting of new classes, methods, buttons and rulers, may encapsulate this material inside a project and load or unload it to GRAPE during run-time.

1.2 Difference to Existing Programs

Dataflow packages like AVS, SGI Explorer and Data Explorer from IBM supply a module library and a network manager to interactively combine modules to a program. Data is modified at each module node and then passed over connections to other nodes. This concept is different to GRAPE's object-oriented mechanism where a current object exists (i. e. the currently marked branch of the data hierarchy) to which methods are sent again and again until a problem is solved. GRAPE's technique is best when analyzing a problem without prior knowledge which sequence of methods could solve it, while the hardwired AVS-style networks are best when work has to be done over and over again in a fixed order.

GRAPE does neither compete with data viewers like Geomview since it offers a mathematical programming environment nor with formula packages like Mathematica which do not contain advanced interactive three dimensional graphics.

Chapter 2

History

This chapter lists the changes between different **GRAPE** versions (from version 5.0 on), especially the ones which require changing existing programs, and the major new features of the versions. For new users this chapter might be less interesting but for those who are using an older **GRAPE** version and want to update

it is important to read this chapter carefully.

We try to keep the new versions as backward compatible as possible, changes which require changing programs are only introduced to solve existing and to avoid future problems.

2.1 Changes and New Features in Version 5.3

New Class Syntax

We have introduced a new syntax for class definitions using `#define`s for the class structures. This allows to easily change classes without the need of changing the subclass definitions and it makes the class definitions more readable.

Since many classes including the base class `Root` have been changed you have to change your own classes to make them work — otherwise using them will crash the program. Please refer to section 5.3.3 for details.

`"delete"` Replaced by `"free"` and `"zero-refs"`

Don't worry, an instance is still deleted by calling the `"delete"` method on it, but to avoid problems with the `refcount` handling the delete mechanism has been changed.

The method `"delete"` now exists only on the base class `Root`, it cannot be defined for other classes (this will result in an error message). Instead of `"delete"` a class now has to define a method `"free"` which is called by `Root "delete"` when the instance is no longer needed and its variables can be freed. The `refcount` now is handled by `Root "delete"` (and `"zero-refs"`, see below).

Converting existing `"delete"` methods to `"free"` methods is easy, just rename the methods from `"delete"` to `"free"`, remove any code that checks or manipulates the `refcount` and call the superclass method `"^free"` instead of `"^delete"` at the end of the method.

`"free"` must neither check nor change the `refcount`, if you really need a special `refcount` handling (this is probably only the case when cyclic references are used as for `TimeStep` and `Geom2d`) you can define a `"zero-refs"` method which is then used by `Root "delete"` to determine if an object can be freed.

For further information and examples see sections 5.4.4.1 and 5.4.5.2.

"list-alloc" and "list-free"

The functionality of "list-alloc" and "list-free" is now identical for all geometry classes. The sizes of the instance lists are changed to exactly the given new sizes therefore this method can be used to enlarge or shrink the sizes of the lists. Existing data is preserved whenever it is possible, in case the lists are enlarged it is copied to the new lists if necessary. To be able to shrink the lists the usage sizes (for Triang classes the number_of variables) have to be set correctly, especially lower than the new list sizes, to indicate that the data is no longer needed the usage sizes can be set to zero before calling "list-alloc". See section 5.4.4.2 for more information.

You only have to change existing programs when you encounter error messages from the changed methods, in this case it should be sufficient to set the usage sizes correctly (they weren't check by some methods in the previous version). Nevertheless you should consider replacing your own code for enlarging or shrinking by calls to "list-alloc", this makes the code more readable and avoids problems with subclasses.

double Instead of float

Almost all float variables in **GRAPE** including the coordinates of the Triang classes have been converted to double. There are two main reasons for this change:

- Higher accuracy. The visualization usually doesn't require the accuracy of double, but numerical problems often do. Therefore many **GRAPE** users used to store their data in double variables, using the **GRAPE** classes only for the visualization.
- Efficiency. Most processors internally use double for floating point computations, float variables have to be converted to double before the computation and back again afterwards. Using double therefore speeds up programs on many systems since the conversions are not required.

Even though the coordinates now need twice as much memory a (large) triangulation only needs about 20–25% more memory since the vertex and neighbour information needs more memory than the coordinates anyway.

Converting a program from float to double should not be too difficult. First you must change all class definitions to the new syntax (see above), this reduces the number of float variables you have to convert. Then replace all float types in your classes, methods and functions (of course only the ones you have to or want to convert) with double and correct the scanf calls to make sure the parameters and the format string still match. Please note that types like VEC3 and MATRIX44 also have been converted to double, for compatibility there are FVEC3 and FVEC4 but you should avoid using them.

If you store data in your own binary format you have to adapt the routines to the new data structures by using temporary float variables for reading and writing (this is the worst case, ASCII formats are obviously no problem and of course we have adapted the **GRAPE** routines). Please consider using the new **GRAPE** XDR archiving system (see below) instead of your own binary format.

New Definitions for Standard Macros

The definitions of standard **GRAPE** macros like END_METHOD, ASSIGN and ASSURE were changed to make them more usable. For example you now can write

```
if(error)
  END_METHOD(NULL);
else
  END_METHOD(self);
```

without getting weird results. Now these macros require the terminating semicolon, you will get error messages from the compiler if they are missing.

Changes to `Adapt2d`

The `Adapt2d` refine flags are no longer cleared by the methods of the **GRAPE** library, this is left to the user's responsibility. Also the boundary functions have been changed, see section 6.3.3.

Naming Changes

The names of all `COMPLEX` functions and constants have been changed to make them compatible with C++, the usual **GRAPE** `g_` prefix has been added to them (compare 10.2.17).

Some constants from the handle also have been renamed, they are listed at the end of section 9.1. For compatibility the old constants are still available (as `#define`s), but they will be removed in one of the next versions.

Faster GRAPE Kernel

The **GRAPE** kernel now is not only more stable but also much faster. The **GRAPE** call, i.e. finding a function for a given method string, was improved by caching the last method called (this speeds up subsequent calls to the same method) and by hashing method strings, see 10.2.10.

Resource File `.graperc`

With the **GRAPE** resource files `~/.graperc` and `./.graperc` which are read when the manager is started you can define keyboard macros for the control window (see 4.3.10) and the default settings for the Postscript driver (see 4.3.11.1).

In fact the `ResourceDev` instances can be used to read any kind of structured user defined data (see 9.3), but at the moment it is only used by the manager and the Postscript driver to read the configuration data from the `.graperc` files.

An example `.graperc` file can be found in the **GRAPE** demo directory.

XDR Archiving System

The new **GRAPE** XDR archiving system provides routines to write data to and read data from files, pipes and sockets. It uses a machine-independent data format — this allows for example to read data on an SGI that was written on a Linux machine and vice versa — and handles references to instances and data arrays.

The system was designed to replace the old `"read"` and `"write"` methods. All write methods used in the manager (e.g. by the `write *.st` and `write *.gr` buttons in the manag menu) use the new routines to write the data, the corresponding read methods are able to read both the old and the new format (the `"read"` methods are still available for backward compatibility).

If you have written `"read"`/`"write"` methods for your own classes you should have no problem writing `"xdr"` methods for them since this is much easier. If you are using your own data format please consider using the XDR format instead.

A detailed documentation of the **GRAPE** XDR archiving system with many examples can be found in section 5.7.

Postscript Driver With Gouraud Shading

The Postscript driver is now capable of creating Postscript files which use Gouraud shading instead of flat shading. Because the shading is carried out by the printer and not by the Postscript driver the files are nevertheless small and resolution independent. The Postscript file contains a parameter "Gouraud accuracy factor" that helps to compromise between printing time and image quality.

The driver now also works faster due to improvements of the intersection tests and needs less memory. For further information refer to 4.3.11.1 where you will also find a description of the new configuration options.

Coordinate Systems

You can now define multiple independent coordinate systems in the scene tree, additionally there is a global coordinate system available in the trans menu.

Coordinate systems are inserted in the scene tree by adding the *CoordSys* project, they can be saved/loaded like other objects with `write gr`/`read gr`.

The coordinate systems are highly configurable, see section 8.3.5 for more information.

Online Help

GRAPE now has a context sensitive online help feature, pressing the control key together with the middle mouse button on an interactive element pops up the associated help text. You can set a help text for your own interactive elements with `"set-help-text"` (see 9.1.3.15) or with the `I_HelpText` item tag (see 5.5.3).

New Utility Functions

- **GRAPE** now has a block memory manager which allows to very efficiently allocate and free large numbers of equally sized small memory blocks, see 10.2.1.2.
- There are new string functions, they should be used instead of the usual C library function to avoid mixing memory allocated with `mallac` and `mem_alloc`, see 10.2.9.
- Class `G_List` has been extended with an iterator stack, this allows to safely use iterators in nested loops, see 10.1.1.1.
- **GRAPE** now provides some machine-independent time functions, see 10.2.13.

Chapter 3

What Basic Concepts are Implemented

This chapter will give you an overview of the basic structures and concepts of **GRAPE**. So both for users and programmers this chapter should be of interest, first of all to understand the main technical foundations of GRAPE, and secondly to give you an idea what already exists and what you can build upon. Unfortunately there are many people who spend a lot of time and energy to implement algorithms which are already present in the GRAPE library. Using all the features of the GRAPE library will save you a lot of time.

In our overview we will already use terms like class and method. GRAPE is an object-oriented environment. In case you have no idea about object-oriented techniques think about a class as a category of things with the same properties and functionality, the features or functions of an object or instance of a class are described as methods. If you want to read more about object-oriented programming please refer to section 5.1.

3.1 Triangulations And Refinement Methods

In GRAPE a triangulation can be created by just calling interactive methods within the manager. There are many methods to manipulate triangulations, techniques like cutting holes, local refinement and merging triangulations are available. So you can create a complex triangulation which then can be used for example as a starting triangulation for some numeric methods. The basic structures for triangulations are the classes called Triang2d and Triang3d for two respectively three dimensional triangulations. There exist many display methods to present your triangulation in the graphics window: you can display the triangulation in both grid and patch mode, special methods allow you for example to display those elements with the spatial angles within two bounds or to display a layer of triangles whose centers lie in a layer surrounding a certain plane. An interpolation method allows you to interpolate between two given triangulations with the same discretization. Besides that there exist functions to read in and to write triangulations for external use. To get a complete overview have a look at the classes in sections 6.3 and 6.4.

Triangulations

- there exist classes for triangulations in two and three dimensions
- triangulations can be created and modified interactively
- modification includes merging two triangulations and cutting holes
- triangulations can be refined globally or locally
- an interpolation between two triangulations can be computed

- triangulations can be displayed in grid and patch mode
- there exist many more special display methods

Working with finite elements you will use the structures FE2d and FE3d. As these classes are derived from Triang2d and Triang3d all methods which are available for the Triang classes can be used. New are some display methods. The method `"isoline-disp"` allows you to display level lines of your finite element structure. `"superlevel"` allows you to display an area with values greater then the given level. There exists an interpolation method that allows you to interpolate between two FExd objects (including the FE data). In case you need local refinement you will need the class Adapt2d. After specifying the triangles which should be refined a new triangulation is produced using this data. A subclass of Adapt2d is Geom2d which is mainly used for visualizing surfaces. If allows to store additional information like normals, colors, parameter lines and boundary curves.

Finite Elements and Geometries

- GRAPE offers classes for finite elements in two and three dimensions
- display methods allow to display level lines and areas an which the values are greater than a given level
- finite elements can be globally can locally refined
- an interpolation between two finite elements can be computed

3.2 Displaying Data Hierarchies

In order to understand the basic concept how the data is displayed in the graphics window you need to know more about the class Scene. An instance of the class scene contains all information that is necessary to present your geometry. Besides a reference to the geometry a scene contains information about the surface properties and positioning of the object. By having a look at the surface and lighting menu of the manager you can easily find out how these different parameters change the appearance of your geometry. You can combine scenes to a linked list or a binary tree. If you want to display a number of copies of your object with different display methods or properties you can make copies (of the scenes, the object doesn't change). A subclass of Scene is Chain, the methods on chain assume to find a linear structure. In case you need time dependent objects please refer to the next section.

Scenes

- an object of the class Scene includes a reference to a geometric object and the necessary information to display it in the graphic window
- a set of objects of the class Scene can be combined to a linked list or binary tree.
- special functionality for linear linked lists are provided

3.3 Time Dependent Structures

GRAPE allows you to express time dependency in your data and thereby to produce animated sequences. There is a time ruler in the manager which lets you change the time interactively. The fundamental idea of the realization of time dependency is to create a class that includes a method "get–object" which can provide your geometry for any given time. One of the already existing classes which provides such a method is the class TimeStep. Its concept is to store a number of objects — e. g. geometries or numerical data — and the timevalues they belong to in one dynamic object. By interpolating these keyframe–geometries, GRAPE can provide a geometric object for any given time and as a consequence produce a continuous animation. An instance of the class TimeStep realizes a linked list that contains the keyframe–geometries. If you use adaptive geometries, you let GRAPE know how the geometry has changed by storing two geometries belonging to the same timevalue: One of these keyframe–geometries is used for interpolating with the previous timestep, the other for interpolating with the next timestep. If you want to make use of another concept of time dependency, you can easily implement that. In some cases it might for example make sense to rather calculate the object for a specific time from the data than use stored objects which need a lot of memory. To implement your own concept of time dependency, you only need to implement a class of your own that provides your own method "get-object". Time–dependent objects become part of the object hierarchy with the help of the class TimeScene, a subclass of the class Scene. In addition to an instance of the class Scene, an instance of the class TimeScene contains the current timevalue and a reference to a dynamic object such as an instance of the class TimeStep.

Time dependent structures

- GRAPE allows time–dependent objects and the production of animated sequences
- creating time–dependent objects by interpolating keyframes is supported by the GRAPE class TimeStep
- GRAPE allows you to implement your own concept of time dependency

3.4 Extending Grape

In case you find the basic classes of GRAPE useful but still need some special methods or new classes, don't hesitate to extent GRAPE so that it fulfills your own requirements. The prerequisites are that you have a working knowledge of C and an idea what object-oriented programming is. By the time the GRAPE project was planned and implemented there was no powerful object-oriented programming language around. So the object-oriented structure was realized in C. When you start programming think of classes as normal structs and of methods as functions. What GRAPE provides is a huge library of classes, methods and utility functions. Whatever you plan to do, you can always make use of objects like buttons, rulers and other interactive elements to be able to work with your application in the interactive GRAPE environment, this way your methods will be easily accessible. In chapter 5 you will find an introduction to programming with **GRAPE**. We hope will help you to get started — all you need is a C-Compiler and some patience.

Extending GRAPE

- GRAPE is a library written in C
- GRAPE provides an object-oriented environment
- Classes and methods can be easily implemented and added to the library
- Interactive elements can be used to access methods and to control parameters

3.5 Projects

By reading through the index you will have realized that a large part of the manual is devoted to describing projects. Projects are the basic concept in GRAPE for organizing the program and enhancing the modularity of the package. A project consists of a set of classes, methods and interactive elements (a “user-interface”) for a specific problem, it can be added and removed interactively during runtime. There are lots of predefined projects, so before you start programming yourself its certainly worth to have a look at them to see what's available — see section 4.5.4 for a short overview, chapter 8 for a detailed description. Using projects is easy, you just have to know how to handle them with the GRAPE manager (see section 4.5.3). For programmers using a project has many advantages, an example can be found in chapter 5.6.

Projects

- help organizing the program by combining classes, methods and an user-interface to a (self-contained) package.
- The predefined projects provide much functionally for solving different problems.
- Projects can be handled easily by the user, they provide a good framework for programmers who want to write their own application.

Chapter 4

How to use a Grape Program

This chapter explains using GRAPE as an application. It refers to the demo program included in the GRAPE package, a guided tour will help to explore all the basic features GRAPE offers.

We presume that GRAPE has been installed successfully. If not, we recommend that paths, the location of the header file and library, and the makefile be checked by your system administrator.

Having compiled and started the included demo program, you should be looking at two GRAPE windows on your screen now.

4.1 Different Types of Interactive Control Elements

Before we start to describe the GRAPE control window in general, we would like to describe the single control elements, called interactives. We start this section with a description of the usage of the mouse buttons in a GRAPE program. After that the different interactives and their usage will be discussed.

4.1.1 The Mouse in GRAPE

GRAPE uses three mouse buttons. The left mouse-button is used to select things. The middle mouse-button is used to abort actions. The right mouse button is used for specific purposes or to open the configuration-menu some interactives provide (this is described in section 4.1.2).

GRAPE distinguishes between two different mouse-actions: "click" and "drag". "click" means that the mouse button is pressed and released while the mouse cursor is located within the to-be-selected control element. "drag" means that the mouse cursor is being moved while the mouse button is kept pressed.

4.1.2 Interactives

4.1.2.1 Layer

To get a more useful and logical menu-structure several parts of the control element may appear in a layer that pops up and is placed above the normal GRAPE-menus. An empty layer (without any control-elements) is shown in figure 4.1.

A layer behaves like a window inside a normal window. It can be moved by dragging its border with the left mouse button, or resized by dragging its border with the right mouse button. If you want to shrink/resize it to its minimal/normal size, you can do this by pressing the left mouse button while the right one is held down and the mouse points at the title line. A shrunk layer is reduced to its headline. Most layers contain a button named either "done", "dismiss" or "close" to remove the layer.

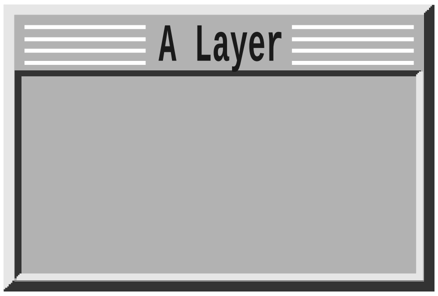


Figure 4.1: Empty Layer

4.1.2.2 Button

Buttons are the most used control-elements in **GRAPE**. They are used for switching between the menus, starting actions and several other things. They can be selected (pressed) with the left mouse button. Inactive buttons cannot be selected (see figure 4.2).

Figure 4.2: Buttons: normal, pressed, inactive

There are some special types of buttons:

Cycle-button: Looking like an ordinary button, this kind of button changes its label every time the button is clicked. Therefore it offers to pick out one out of several options.

ComboButton: Serving the same purpose as a Cycle-button, ComboButtons show all possible selections in a list when the left mouse is hold down. An entry is selected by releasing the mouse-button.

Checkbox: Checkboxes turn switches on or off (see figure 4.3).

Figure 4.3: Checkboxes: not pressed, pressed

4.1.2.3 Inputs

Inputs are fields where text or numbers can be entered. As long as they are not selected, they look like ordinary buttons. After they have been selected, new text can be typed in to replace an existent entry, or an already existent entry can be edited by starting with one of the following keys: "cursor left", "cursor right", "home" or "end" (see figure 4.4).

Figure 4.4: Inputs: pressed, replacing an entry, editing an entry

Press "enter" or the left mouse button to finish typing, use the middle mouse button or select another input to restore the contents of the input field.

Reference of special keys:

cursor left/right	move the cursor one character backwards/forward
home/end	move the cursor to begin or end of line
delete	delete a character at cursor position
backspace	delete a character backwards
escape	clear the input field
enter	end typing
tab	either complete the current word, or switch to the next input field of the same layer (depends on the context)

4.1.2.4 Slider

A slider (see figure 4.5) is used to adjust values in a given range, e.g. percentages.

Figure 4.5: Slider

There a two possibilities to set the slider to a new value: One is to use the input field on the right to type in the new value, the other is to click or drag the bar to a new position using the left mouse button, while the input field is being updated continously.

The right mouse button makes a configuration layer (see figure 4.6) pop up.

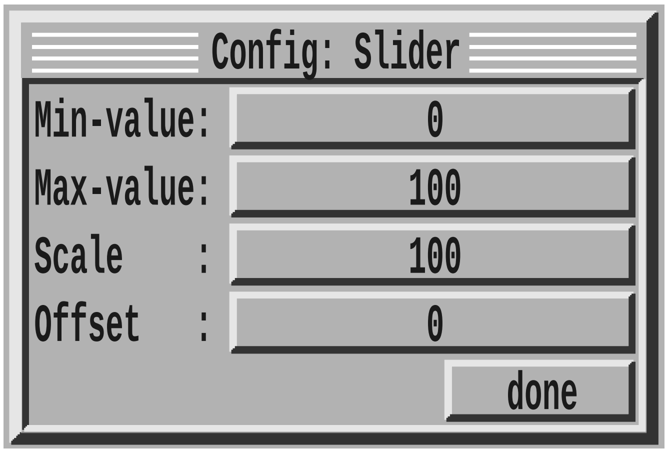


Figure 4.6: Slider: configuration layer

The Min-/Max-values are the minimal and maximal values the slider can be set to. Scale is the range the slider shows from its left to its right position. Offset is the value shown when the slider is set to its left position (that implies that the value shown when the slider is set to its right position is the sum of Offset and Scale).

4.1.2.5 Ruler

A ruler (see figure 4.7) is used to control various values.

A ruler is used just like a slider. The additional buttons (arrow left/right) set the value step by step, if pressed more than half a second, repeatedly. The step size can be changed in the configuration layer of the ruler (see figure 4.8).

Figure 4.7: Ruler

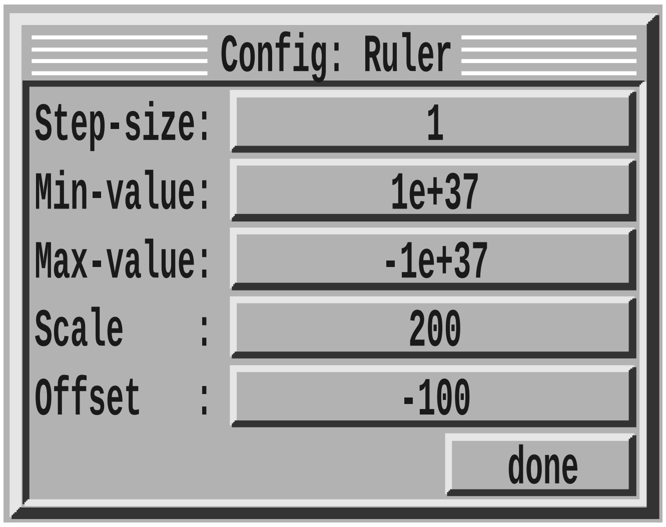


Figure 4.8: Ruler: configuration layer

4.1.2.6 Time-slider and Time-ruler

There are also special kinds of sliders and rulers which can be time-dependent, i.e. dependent on the main time ruler of the manager menu (see 4.3.5.2), or – more generally – depend on any other variable. They are called function-slider or function-ruler and consist of an additional button on their left, named $\boxed{\text{E}}$ (see figure 4.9).

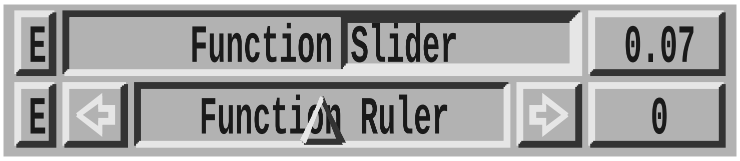


Figure 4.9: Function-slider and function-ruler

This button pops up a spline-editor (see 4.1.2.10) that offers to change the time-dependency function and to switch the dependency on and off. If the dependency is on, this button is drawn pressed down, and the slider/ruler changes its value by evaluating the spline with respect to the current value of the main time ruler.

4.1.2.7 Scrollbar

Another control element similar to the ruler is the scrollbar (see figure 4.10).

A scrollbar is often used to scroll a list of file names or method names. The size of the bar corresponds to the proportion of visible and invisible entries. By scrolling the bar up and down – continously by dragging, linewise by using the step buttons, or pagewise by clicking above or below the bar –, the currently visible part of the list can be changed.

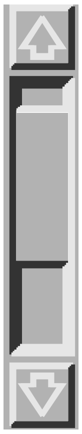


Figure 4.10: Scrollbar

4.1.2.8 List

A list (see figure 4.11) is designed to select one out of of several possibilities, like the names of files or methods.

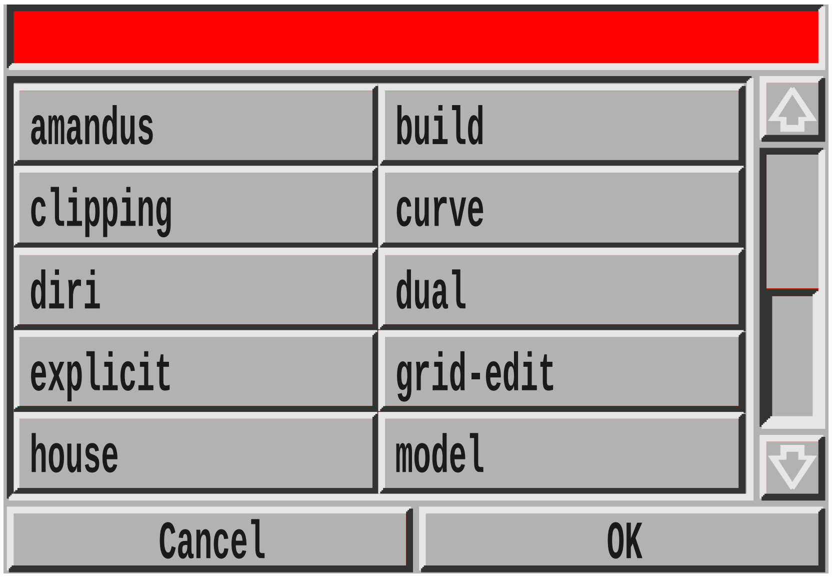


Figure 4.11: List

The top line of the list is an input in which an item can be selected by typing its name. The tabulator key can be used to complete the name.

On the right is a scrollbar which is used to scroll the list. To select an item of the list, click it with the left mouse button and then click `Ok`, or just click the item with the left mouse button twice. Using the right mouse button in the list of items, makes the list scroll up or down. The direction of scrolling depends on your position inside the list.

`cancel` is used to cancel the selection and quit the list.

4.1.2.9 FileList

A `FileList` (see figure 4.12) is a list in which file names can be picked out. Enter path and wildcard in the upper input (eg. `"/home/me/*.st;*.gr;grape*"`). The tab key completes the file name with respect to the working directory.

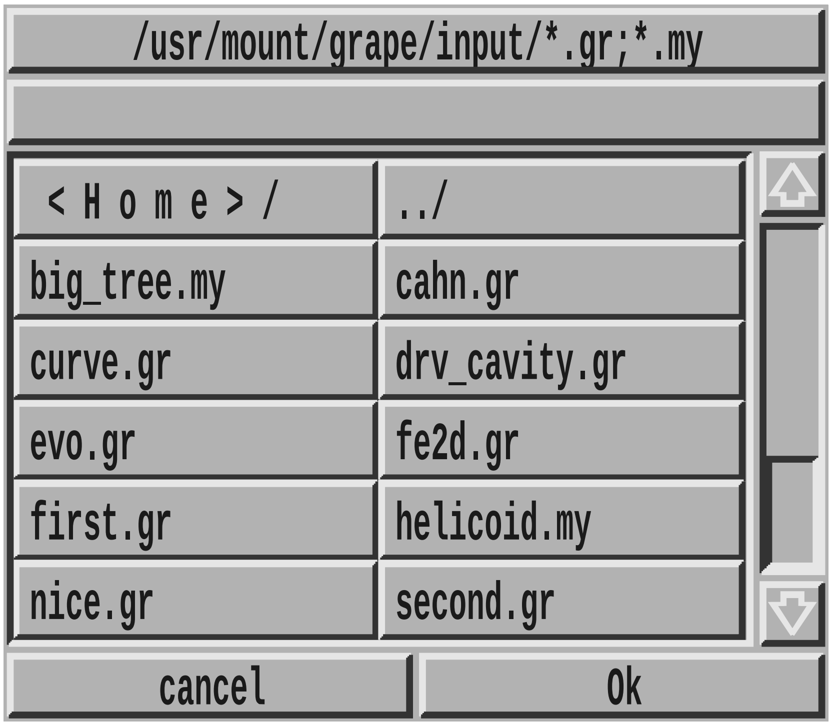


Figure 4.12: FileList

4.1.2.10 Spline-Editor

The spline-editor (see figure 4.13) is used to define the time-dependency function of time-dependent sliders and rulers.

You can change the spline by simply dragging its points or the tangent of the point to the new value you like. Another possibility is, to select a point with a click, and enter its new coordinates in the inputs named `x` and `y`. To add new points to the spline, simply click at the chosen position with the right mouse button. Remove points by keeping the shift key pressed while clicking the right mouse button.

`Done`	close the spline-editor
`Use`	switch the time-dependency of the ruler on or off
`?`	show a brief help for the mouse button usage in the spline-editor
`FixX`/`FixY`	limit the way spline-points can be moved – `FixX`: only y-direction can be changed, `FixY` only x-direction can be changed
`del`	delete the current spline
`load`/`save`	load/save the spline to disk
`resc`	rescale the spline-editor so that the complete spline is visible
`x:`/`y:`	show/set the coordinates
`zoom`	zoom (reset with rescale button)

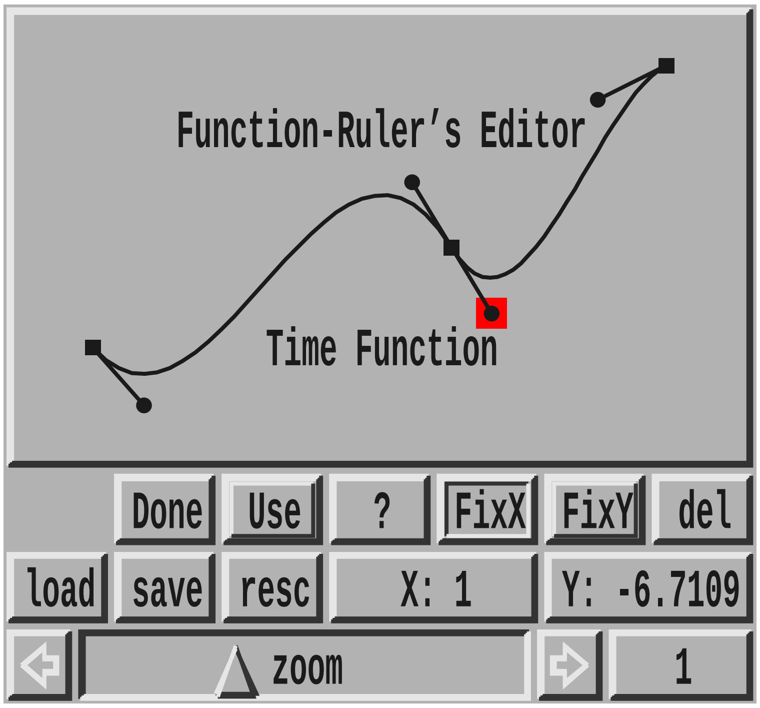


Figure 4.13: Spline-Editor

4.2 Manager and Graphic Device

There are two machine dependent devices in GRAPE, one for 3D graphics and one for control elements. With this concept its is possible to use GRAPE on a large number of different systems and with different kinds of 3D graphics hardware.

4.2.1 Graphic Device

The most important device is the graphic device, which enables you to view your mathematical objects. The typical output of a graphic device is the graphic window an interactive GRAPE program starts with. The graphic device has functions to draw different type of patches, lines and some predefined objects, all in 3D-coordinates. The device will adapt these functions to your machines hardware possibilities. For example the postscript output of GRAPE is also realized as a graphic device. For a detailed description of the graphic device see 9.2.2.

4.2.2 Control Device

The control device is used by the manager to show its menus and get the user events. The actual GRAPE version only has an X11 control device because all machines supported up to now do support X11. The manager also controls the graphic device, with the manager you can change the preferences of the actual graphic device like grid or patch mode, doublebuffer and several other things. The manager can never be reviewed without the graphic device because a lot of actions made in the manager only manipulate the graphic device, for example the rotation of an object is done by changing the rotation matrix of the graphic device.

The manager itself and its menus are described in the next section.

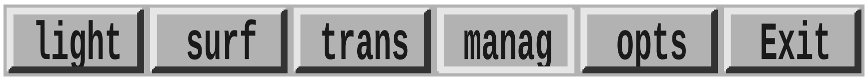
4.3 Controlling and Viewing

In GRAPE, a basic set of interactive features is available independent of the projects loaded; in this chapter we will introduce all of these standard features. They include operations on geometrical objects like translating or rotating in space, changing surface shading colors, but also creating light sources and maintaining the object hierarchy. For short, we could say, the standard operations divide into viewing operations and controlling operations.

In this chapter we will give a detailed description of the different elements in each menu. Make sure that you understand the **GRAPE** displaying concept (see 4.3.5). Only the knowlege of this basic concept will enable you to understand what's going on.

4.3.1 The Five GRAPE-Menus

Most of the time, you will interact with GRAPE through the control panel. Have a close look at it: at the top there's a row of buttons which allow switching between the five menus (and an Exit button to leave the program):



This row of buttons will always stay there; it is common for all menus. The concept of menus mainly serves the purpose of saving space in the control panel; imagine a control panel crowded with all interactive objects at once!

Located at the bottom of the control-window are a copyright-message and a message-field with up to three lines for general use. Projects display their name, author and version string here, some routines print out a special use of the mouse-buttons, some other methods display their current or termination state. At startup it should contain the current GRAPE-version-string.

Pressing the `?` button will give you an advice how to get context sensitive help where the programmers found it helpful: simply press Control-MiddleMouseButton over a place you want to get help for; after having read the message press `dismiss` to close the help window.

The menus are designed to fulfill different tasks:

`light` Create and manipulate light sources illuminating the scenery. Their colors, directions, and positions can be influenced.

`surf` Edit surface properties. Surface properties define coloring and shading of surfaces.

`trans` Move, scale or rotate the scenery; this affects the current transformation matrix.

`manag` Managing facilities, which includes many general operations. This menu is automatically selected when you have started GRAPE.

`opts` Options menus. The project-specific environment is built up here.

`Exit` Exit the event-loop (normally to exit the program).

4.3.2 General Interaction Strategy

While you are working, GRAPE keeps two things in mind: the current collection of visible objects and the current way of displaying them. In fact, we have the concept of a current object rather than a collection of objects, but the whole branch below is displayed and manipulated in a way. As explained below, the current object can be set via the hierarchy view in the manager menu. Every time you perform some action — like clicking on a button in the control panel or shifting a ruler — that could have changed the graphic view GRAPE redraws the current object in the graphics window using the current display method.

In some cases, especially when your objects are complex, you will find this behaviour inappropriate, because redrawing the graphics every time might be quite slow; then you can disable the redrawing mechanism by `switch` "`draw on/off`" in the manager menu. Another way to save time is by selecting grid style instead of patch style.

In the following, we want to explain the menus and their options in more detail.

4.3.3 Get the Feeling

Have a guided tour through the manag menu in this chapter.

You have created your first executable as shown in section 4. Start it from the command prompt (if is already running and you might have changed something press `Exit` (in the upper right corner) and start it again.)

startup	You should see a graphics window showing a house with a little (hard to recognize) tree and a control window showing the manag–menu. Depending on the GraphicDevice (X or GL or ...) the house will be displayed as a wireframe or solid by default. (This can be switched, but not now). First have a look into the transformation menu.
`trans`	In the trans-menu. Play with the translation rulers
`trans x`..`trans z`	to move the object, with the
`scale`	ruler to scale it and with
`rot x`..`rot z`	to rotate it.
`reset`	Press reset to reset the transformation if you lost the object.
`sphere`	Switch to sphere rotation mode. You can click inside the sphere and then drag the mouse to rotate the object as if turning a ball.
`ruler`	Switch back to ruler rotation mode.
`reset`	Press reset to reset the transformation.
`use`	Switch the camera on. The house is displayed perspectively.
`use`	Switch the camera off.
`reset`	Press reset to reset the transformation.
`manag`	Back to the manag menu. First we explore the data hiarchy tree (scene tree). In the upper middle of the control window is a button `"main" Scene`, this is the current object. The graphic window's contents are the graphical view of this Scene. Since it is a Scene it does nothing but display its sub node `"House" Scene`. Press this button now:
`"House" Scene`	You stepped into the Scene tree. `"House"` is the current object now. `"main"` moved to the left, ready to bring you out again. Two new buttons appeared, `"Chimney" Scene` and `"Tree" TimeScene`. These are sub objects of `"House"`. Nothing should have changed in the graphics window.

When asked to display, `"House"` will display both Scenes, the tree and the chimney. `"Chimney"` displays the whole house. To see the chimney press

`"Chimney" Scene`	The tree disappears. The sub objects of `Chimney` are the real chimney object `"Chimney" Adapt2d` and the next scene `"Roof" Scene`.
`"Roof" Scene`	The chimney disappears out of the graphics window. Note that the button `"Chimney" Scene` moves up.
`"Walls" Scene`	Only the walls remain.
`"Roof" Scene`	
`"Chimney" Scene`	Return.
`"Chimney" Adapt2d`	Walk onto the object. No sub objects are displayed since there are none.
`"Chimney" Adapt2d`	Press the current object's button. You get an info list in a layer which shows basic information, scroll through the list using the scrollbar at the right side. This info list works for every object. Sometimes you will open it accidently, try to close it by pressing the
`Ok`	button at the bottom of the layer.
`mark`	We now want a second chimney. Mark the existing one. The button label changes, the marked object's button becomes blue.
`"Chimney" Scene`	We need a place to copy the object to. This must be a Scene. The same scene is a good place.
`copy`	Link the object to another place. A Scene "C: Chimney" is created. You don't see the copy because the objects are drawn at the same place.
`"C: Chimney" Scene`	Go there. The scene's object is marked, too. It is the same object controlled by two scenes. If you had pressed `hardcopy` you would have created a new object.
`unmark`	Remove the marking.
`scene-trans`	Open the scene transformation menu. The details are described in the trans-menu. The rulers are grayed out because there is no active matrix.
`no matrix`	Press and hold the combobutton to select `model`. It always shows the current state.
`trans x`(`←`), twice	Move the second chimney in direction `-x`. The ruler's input field shows `-0.4`.
`dismiss`	Close the scene matrix editor.
`"Chimney" Scene`	Walk up to see both chimneys.
`scene-suprop`	Open scene surface property editor to switch to grid mode to see the triangulation. Doesn't hurt if grid mode is already active.
`to grid`/`to patch`	Click this button on the scene suprop layer until it shows "to patch" indicating the local scene surface property is in grid mode.

`"C: Chimney" Scene`	Note the surface property layer has updated properly. This Scene doesn't have an own surface property and is displayed using the global surface properties. Automatically create a surface property by pressing
`to grid`/`to patch`	Until it shows `to patch`.
`dismiss`	Close the scene surface property editor.
	We do not want to walk on the two other Scenes by hand, let's use a utility method.
`send method`	We want to send the method `"all2grid"`.
`all2grid`	Pick the right method in the list.
`all2grid`	Click the entry again, that is another way to press `Ok`.
	The roof and the walls should be in grid mode now.
`"Chimney" Scene`	Walk up.
`"Chimney" Adapt2d`	Refine the chimney now.
`send method`	We want to send the method `"refine-global"`. This is a method of class `Triang2d`. Because we stay on an instance of `Adapt2d`, which is subclass of `Fe2d`, which is subclass of `Triang2d`, we have to change to the `Triang2d` methods first.
`../ (Fe2d)`	Click the path to the superclass methods.
`../ (Fe2d)`	Click again to enter the decision.
`../ (Triang2d)`	Click the path to the superclass methods.
`../ (Triang2d)`	Click again to enter the decision. Use the scrollbar at the right side to find the string `"refine4"`.
`refine4`	Pick the right method in the list.
`refine4`	Click the entry again.
`"Chimney" Scene`	Go back to the scene. Note that both chimneys are refined since they are the same object.
`"C: Chimney" Scene`	We want to remove the extra chimney. Go to the Scene.
`delete-one`	Delete the Scene and its object but not the next Scene. Note that the original chimney remains in the Scene.
	You stay on `"Roof" Scene` now. Refine the rest of the house as well by using the Scene.
`send method`	You see the valid methods of the class Scene. But Scenes are special, if a Scene does not understand a method it is sent to the sub objects.
`"refine4"`	Enter the text in the input field and press ENTER.
	Both, the roof and the walls are refined.
`mark`	
`copy`	

`unmark`	Create a copy of the roof.
`"C: Roof" Scene`	Go to the new copied Scene.
`scene display method`	Change the display method only for the roof, which is of class Fe2d. A list in a layer appears, showing the valid display methods for Fe2d.
`isoline`	
`Ok`	Choose isoline as display method. The roof is displayed as "temperature isolines".
`"Roof" Scene`	See one roof with two display methods.
`"C: Roof" Scene`	Go to the copy of the roof.
`delete-one`	Delete this Scene and its object.
`"Roof" Scene`	
`"Chimney" Scene`	
`"House" Scene`	
`"Tree" TimeScene`	Go to the tree to learn something about time dependent objects.
`"interpol" Triang2d`	Walk onto the interpolated object. The object moves to center position in the graphics window because the TimeScene has an own Scene matrix that is unused now.
`"Tree" TimeScene`	Go back onto the TimeScene.
`TimeScene:  "Tree"` ruler	Move the scene time ruler. See the tree growing and shrinking. Note that tree segments appear and disappear, this is done by several TimeSteps.
`"t= 0.000" TimeStep`	Go to the chained TimeSteps that are interpolated. The Triang2ds on the right side are the pre-object and the post-object.
`"t= 0.100" TimeStep`	Visit the next TimeStep.
`"t= 0.200" TimeStep`	Another TimeStep yet.
`"Tree" TimeScene`	Go back onto the TimeScene. You do not have to go via the prior TimeSteps.
`main time`	Move the main time ruler. Nothing happens.
`TimeScene:  "Tree"` (`I`)	Initialize the time dependency spline.
`TimeScene:  "Tree"` (`E`)	Open the Spline_Editor to edit the scene time spline, which describes the main time dependency. Do not edit the spline now, just press
`Use`	to use the spline. Note the `E` is pressed now. Then press
`Done`	to close the Spline_Editor.
`main time`	Move the main time ruler. The scene time ruler moves as well. Note that only main time values between `0` and `1` are valid.
`TimeScene:  "Tree"` (`E`)	Open the Spline_Editor to edit the scene time spline. Move the little circle at the right side of the left node up to get an `S` shaped spline.

`main time`	Move the main time ruler. The scene time ruler moves as well, but not linear any more. Note you do not have to close the Spline_Editor to see the effect. To switch off the spline and to close the editor, press
`Use`	
`Done`	
`"House" Scene`	
`"Chimney" Scene`	Let's try an alternative display method.
`display method`	Open a display method selector layer.
`"points" ENTER`	Enter the word "points". The house is displayed as points.
`opts`	Switch to the opts-menu. The display method has put a ruler there.
`ball radius`	Play with the ruler to adjust the points size.
`manag`	Switch back to the manag menu.
`display method`	Reset the display method.
`display`	Click on the default value: `display`.
`Ok`	Accept the value. You can have a look into the opts menu to see that the ruler disappeared.

Now you have seen something of GRAPE; you may try to add and remove some projects now and see their Buttons and Rulers come and go in the option-menus; for the Projects and their user interfaces see chapter 4.5.

The next sections describe everything in detail.

4.3.4 The "Manager" Menu

The manager menu is active after starting GRAPE. From here you can organize your work by loading projects, you can walk through the scene hierarchy and organize it, and apply methods from the method pool. Also reading and writing certain data objects, or switching various flags is done here.

4.3.5 Data Hierarchy Tree and Displaying Concept

The cross-shaped arrangement of buttons in the middle of the manager's menus represents and controls a hierarchy of objects. All objects are associated with a node in a (partly seen) binary tree. Each button belongs to a node of the object hierarchy, in particular the middle button shows the current position in the tree. Clicking this button supplies some information about the present object, which information it provides is dependant of its class. Clicking one of the neighbouring buttons changes the current position. The button on the left side is the next upper object where you come from, the direction to the tree's root, buttons above and below the current object represent associated objects at the same level. The buttons on the right side are sub-objects. The buttons only a local view of the data hierarchy tree, all displayed objects are direct neighbours of the middle one, especially the buttons in the corners are not neighbour's neighbours, so don't be harmed if objects seem to disappear.

When walking through the tree, you will also see the graphics window change its contents, because it displays the current object (and the objects below). This is done by sending the current display method to the current object. Scenes automatically send the method not only to its object,

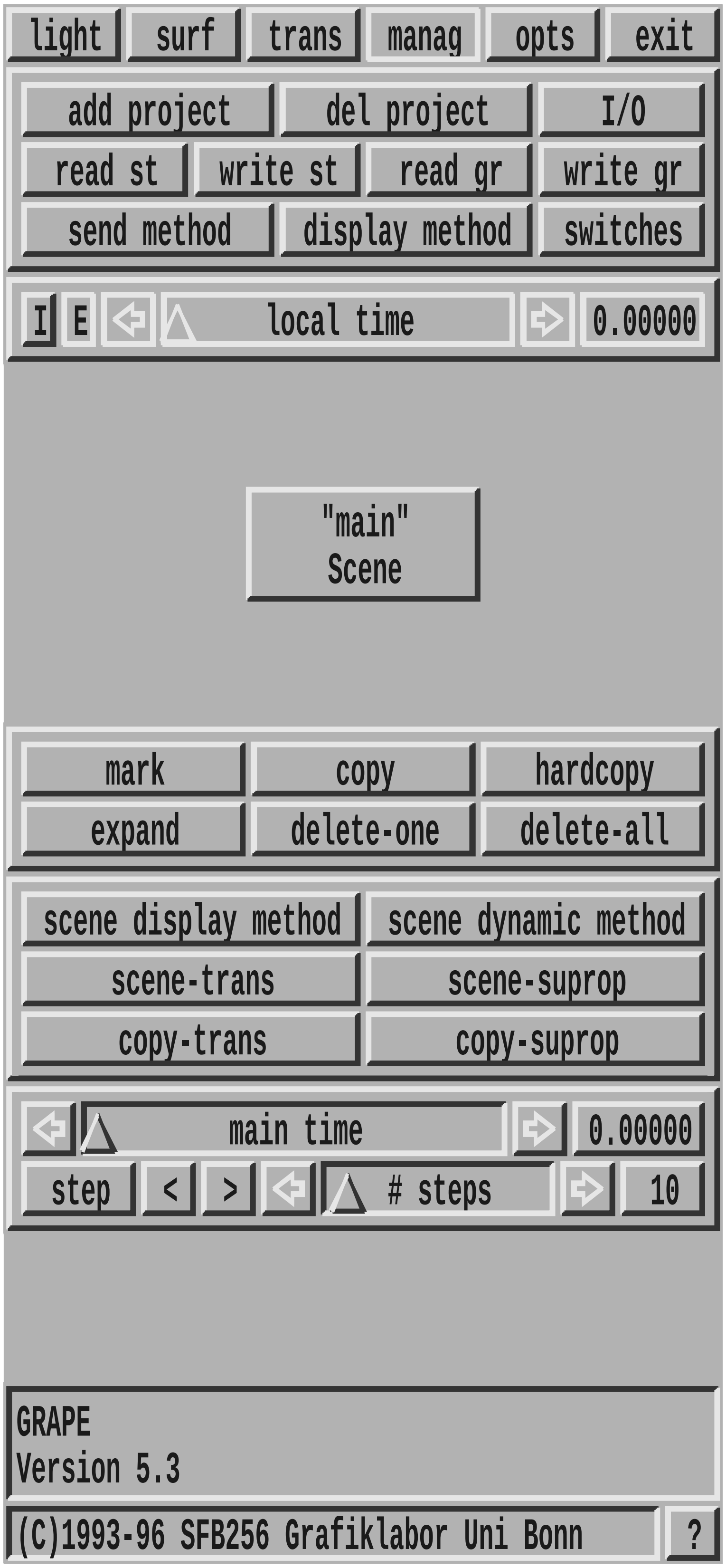


Figure 4.14: Manager Menu

but also to the subscene. A scene is also capable of sending a different (own) display method to its object and/or changed graphic properties, e.g. grid/patch switching or object transformations, locally for its object. Using this mechanism and different scenes with the same object (see further below) one object can be drawn using different display method at the same time, e.g. isolines and grid of a finite element solution.

4.3.5.1 General Functions

`add project` This button pops up a list of available projects. Choose one to add out of a collection of predefined projects, i.e. to add its classes, methods, interactives in the opt-menu and initial entries for the scene-tree. The existing projects are described in chapter Projects 8.

`del project` This button pops up a list of added projects. Choose one to remove its user-interface part and its methods. The other things like classes cannot be removed savely. Delete one of the previously added projects. This will free your options menu of no longer used buttons and rulers.

`I/O` Some export formats. This button opens a layer with buttons for every available output-format. It will be visible in every menu. Normally you will press PostScript to save the current contents of the graphics window as a postscript file — a window with various options will appear, press `Go` to start (see section 4.3.11.1 for a detailed description of the PostScript config layer).

There are also platform-specific output-options for post-processing tools like renderers and ray-tracing-programs. Some drivers need machine licences.

`read st` Read a graphic-STate file `*.st` into the graphicdevice (see below). Unfortunately most of the rulers and sliders will control the relative position (because tha absolute values cannot be determined), so they will be set to middle position.

`write st` Write the current graphic-STate as a file `*.st`. It consists of the current state of the light, surf, and trans menu (the camera itself will be saved too, but the `use` will not show, that the camera is active. If the default coordinate system has been configured, it will be saved and restored either. If you create a new file you should add the extension .st to the filename.

`read gr`, `write gr` Select a grape file `*.gr` and read or write objects in the new GRAPE XDR archiving format (which is machine-independent, is able to handle references and can easily be extended for new classes, see 5.7). Data files in the old format can still be read.

`send method` Send a method to the current object by typing the method's ASCII name or choosing from a list. The list contains all methods which make sense to be sent to the current object. Many methods (without arguments) are designed as user-sendable methods but are not worthy enough to have an own button. (For programmers: these are the methods with the suffix "`-send`".) You can walk through the hierarchy of all superclasses since their methods might also be useful.

`display method` Replace the standard display method **"display"**, so that any change of control will cause the sending of the new method to the current object. As above you can walk through the class hierarchy. Some display methods have own interactives, which will be added to the option menus. These interactives will automatically be removed, if the specific display method is no longer in use. As mentioned above, you need to understand the **GRAPE** displaying concept (refer 4.3.5) to understand this selection. (For programmers: all methods with the suffix "**-disp**" are listed.)

`switches` Open a persisting layer with several switches.

- `to grid/to patch` Toggle between grid style and patch style. With *grid style* active, no filled polygons are drawn, but only their outlines. This speeds up the display process very much and is therefore recommended when working with objects that contain many polygons. In *patch style*, filled polygons are drawn; around lines (Triang1d's) a pipe is drawn, and points are displayed as spheres. If you want the lighting to take effect, you need to select patch style.
- `drawing` A Checkbox: suppress the graphic output to save time.
- `doublebuffer` Use this Checkbox to switch the GraphicDevice between single-buffer mode and double-buffer-mode. If supported, double-buffer-mode suppresses flickering but sometimes has less colors.
- `depthcue` toggle depth-cueing. With depth-cueing enabled, lines drawn in grid style seem like "vanishing in the depth", because the parts further away are drawn with darker colour. Not every machine is capable of depth-cueing, however. If GRAPE seems to have problems drawing lines, try to switch depth-cueing off. In general, if depth-cueing and light-model (i.e. patch mode) are both active the results will be somewhat unpredictable.
- `draw-once` Draw the graphic window contents once, useful if you have switched off `drawing` and want to see the current state.
- `full screen` full screen graphics: the graphics window is enlarged to full screen size. To leave this mode, just press the middle mouse button. Full screen viewing is most useful for making photographs of your geometry.
- `dismiss` close the layer.

4.3.5.2 Walking Through the Time

If your scene contains animated objects, you surely will want to walk through the time to see different stages of your scene. The time is given as a real number between 0 and 1.

`local time` functionruler Using this ruler you can set the local time for the current time dependent object, e.g. a TimeScene. When a current time dependent object is found the ruler's label will show its class and name. The spline describes the main time dependency, click the `I` to initialize the spline, i. e. to read the min/max values from the object and set the spline linear. If no time dependend object is found in the current scene tree, the ruler will be inactive. Otherwise the name and type of the object, controlled by the ruler will be used as the ruler's label.

`main time` ruler — Using this ruler you can walk to a time between 0 and 1. Time dependend rulers, e.g. the rotation rulers in the trans-menu, change their values automatically.

`step` — Selection button to choose the behaviour of the direction buttons `<` and `>`. The label of the button shows the current mode. The different modes are:

step default mode. With *step* active, the direction buttons advance the time step by step.

loop play mode with loop option. The direction buttons act as *play* and *reverse play*. When a bound of the time interval is passed, the animation starts over at the other one, thus looping through the animation. All interactives will get inactive, except a newly created wait layer, showing the current main time. To stop the animation use the `abort` button.

bounce play mode with bounce option. This is the same as the loop mode, but when a time interval bound is reached, the play direction is reversed.

floop fullscreen loop mode. Same as the loop mode, but first the fullscreen mode is activated. The middle mouse button is used to stop the motion.

`<`, `>` — Walk through the time in discrete steps if the animation mode is in *step*-mode. Otherwise start the animation in *play* or *reverse play* direction. The total number of steps is given by the following ruler:

`# steps` ruler — Controls the number of subdivisions of the time interval $[0, 1]$. This is the number or frames of one animation cycle.

4.3.5.3 Hierarchy Management

The data hierarchy tree and its displaying concept was described previous in this section 4.3.5. Now the interactives for building up a scene tree will be described.

`mark` — Mark an object in the scene tree (data hierarchy tree) for operations with two operands, e. g. for copying.

`copy` — Copy the marked instance. Can be activated only if you stay on a scene node. A new scene below the current scene will be included whose object is a reference to the marked instance. Since all sub-objects are part of the marked instance the whole branch is copied.

This type of copy is preferable to the following "real" copy if the same object shall be displayed for example by using different display methods. Then you make a few of these "soft"-copies and change the display method of the corresponding scene nodes.

`hardcopy` — Hardcopy the marked branch. Can be activated only on a scene node. A new scene below the current scene will be included whose object is a real copy of the marked instance.

`expand` — Include a new scene node at the present position.

`delete-one` — Delete the current object. If it is a scene node only delete its right branch (object branch), the down branch (next branch) moves to the current position. If the marked instance is deleted it will remain alive as marked instance, as a target for further copy operations.

`delete-all` Delete the current object. If it is a scene node delete its two branches. If the marked instance is deleted it will remain alive as marked instance, as a target for further copy operations.

4.3.5.4 Manipulating Scene Nodes

As briefly mentioned above a scene node contains a transformation matrix, surface properties, and a display method of its own, allowing to arrange objects independently from others, or to display objects with different material structure simultaneously. (Scene: see section 7.1.1) The separate display method allows to replace the standard method "display" not only globally, but also locally.

`scene display method` Replace the display method for the scenes' object with a new display method. This is stored in the present scene node.

`scene dynamic method` Only works on TimeScenes (see section 7.1.3). Replace the dynamic method for the timescenes' object with a new one. This is stored in the present timescene node.

`scene-trans` Open a layer with transformation matrix editing facilities. The buttons not described here are the same as in the trans-menu.

- `reset` Reset the transformation matrix of the current scene.
- `no matrix/model/view/proj` Choose one of the three matrix modes projection, viewing and modelling. "no matrix" means the matrix is unused. The *mode* and *view* transformations will be applied before the general transformation is done. Only the *proj* trnaformation is done behind the general trnasformation, but also after the light calculation.
- `dismiss` Close the layer.

`copy-trans` Copy the transformation matrix from a marked scene to the current object if it is a scene.

`scene-suprop` Open a layer with surface property editing facilities. See surf-menu (4.3.7).

- `emission`, `ambient`, `diffuse`, `specular`, `linecolor` These buttons appear in the currently set color for the surface properties. Click them to open a color editor.
- `special` Click this button to open a layer with some special value editor sliders.
- `SGl enable` Choose the combination of capital letters to enable the following parts of the scene surface property, making the scene transparent for the disabled modes.

  S surface property: emission, ambient, diffuse, specular

  G grid/patch

L linecolor

to patch/to grid	Switch the local grid/patch mode (see switches) of the current scene. On some GraphicDevices you have to select patch as global value to see mixed grid and patch scenes.
read pr,write pr	Load or save the surface properties as a file `*.pr`. These are compatible with the global files from the surface menu.
from global,to global	Copy the local surface property from/to the global one.
reset	Reset, i. e. delete, the surface properties of the current scene.
dismiss	Close the layer.

`copy-suprop` Copy the surface properties from a marked scene to the current object if it is a scene.

4.3.6 The "Light" Menu

An important aspect of visualization is the use of color and lightning for 3D Effects. That's why GRAPE offers the user this feature with up to nine light sources. The number of different colours and available lightsources depends on the graphics environment. GRAPE supports direction and position lightsources. There is no shadow, so inner parts of surfaces wont't be dark, the best way to see holes is to put a position lightsource inside of the object.

4.3.6.1 The lightsources

0, ..., 8	These buttons represent the different lightsources. If the specific lightsource is enabled the color of the specific number button is yellow otherwise it's gray, e.g. in default configuration lighsource 0 and 1 are enabled. In the colored rectangle below the number button the color of the lightsource can be seen. The user can choose a lightsource by pushing its number button, so that its status is *active*. Then all other buttons are effecting only the behaviour of the active lightsource. If the specific lightsource is enabled the color of the specific lighsource The color of the button is yellow if
on/off	This button is used to switch the active lightsource `on` or `off`.
position, direction	Choose one of the two different types of lightsource for the choosen source: `position` and the `direction`. Position lightsources are point lightsources at a given position, direction lightsources simulate a lightsource at infinite distance.
show lights	This checkbox decides if symbols for the lightsources are drawn. Position lightsources will be draw as an icosaeder, the direction lightsource as an arrow. Using this the changing of lightsources is much easier.
light-color	Compose a colour for the active lightsource with this color editor. (about color editors see Surface menu (4.3.7)). Edit the position/direction with the same Interactives as in the transmenu (4.3.8). Note that the position and distance rulers disappear when editing direction lightsources.

Figure 4.15: Light Menu

`light intensity factor` Functionslider for intensity scale factor for all lightsources. Useful for fading effects in animated scenes.

4.3.7 The "Surface" Menu

The global surface properties represent the material structure and colouring used for all objects that are shown in patch style and do not have their own surface properties.

The rendering of the graphics is done using the Phong light model (but normally with flat-shading or Gouraud-shading). You may edit all the material-dependent parameters here while the light-sources are edited in the light-menu. This menu allows to change the global values; each scene can have local surface-properties which are edited in the manag-menu.

Note that some machines do not have the ability to reproduce the colours in sufficient resolution; you will see the gradation of colours on an object, which has a bad effect on the degree of reality.

The colors are edited with the GRAPE standard color-editor which supports both RGB and HSV model, HSV is some sort of polar coordinates for the RGB color cube, switch the color model by pressing the `HSV/RGB`-Button. The current color is displayed in a rectangle. Note that ambient, diffuse and specular reflection are scaled with the lightsource's color: if their values are blue and the lights are green the result is black. If the diffuse reflection is 50one lightsource with 50is 50

`read pr`,`write pr` these buttons open file selectors which give you the uniue opportunity to load and save SUrface PROPerties.

`reset` resets the surface property to the default values but not background and linecolor, the button will be inactive if there is nothing to reset. Background and line color will not be reseted.

`emission` These rulers control the object's *emission* colour. Increasing the component values makes the surface more and more look like glowing.

`ambient` The *ambient* colour controls the appearance of the surface in shadowy regions. Remeber again, that no real shadows or reflections are used so shadowy means in a bad angle to the light sources.

`diffuse` The *diffuse* colour influences those parts of the surface that are front-faced to any light source.

`specular` The *specular* colour effects the highlights on the surface, namely the small spots where the light sources shine very directly on it. The highlights' size is also controlled by the shinyness parameter:

`opacity` An object's *opacity* value determines how much can be seen through it. The default value is 1.0, which means that the object is fully opaque, whereas a value of 0.0 indicates complete transparency. In simple situations you can achieve reasonable results using transparency, but be careful using this ruler, because current graphics hardware is not fully capable of rendering complex scenes containing transparent objects.

`shinyness` Values for this parameter range from 0 to 59. It describes the inverse size of the highlights.

`background`,`linecolor`,`textcolor` these buttons open additional color editors for the global background, linecolor and textcolor settings respectively. Press `dismiss` to close them.

`invert` invert the global background, linecolor and textcolor settings: use this feature to get a good preview and appropriate settings for PostScript output, where the background will be white paper.

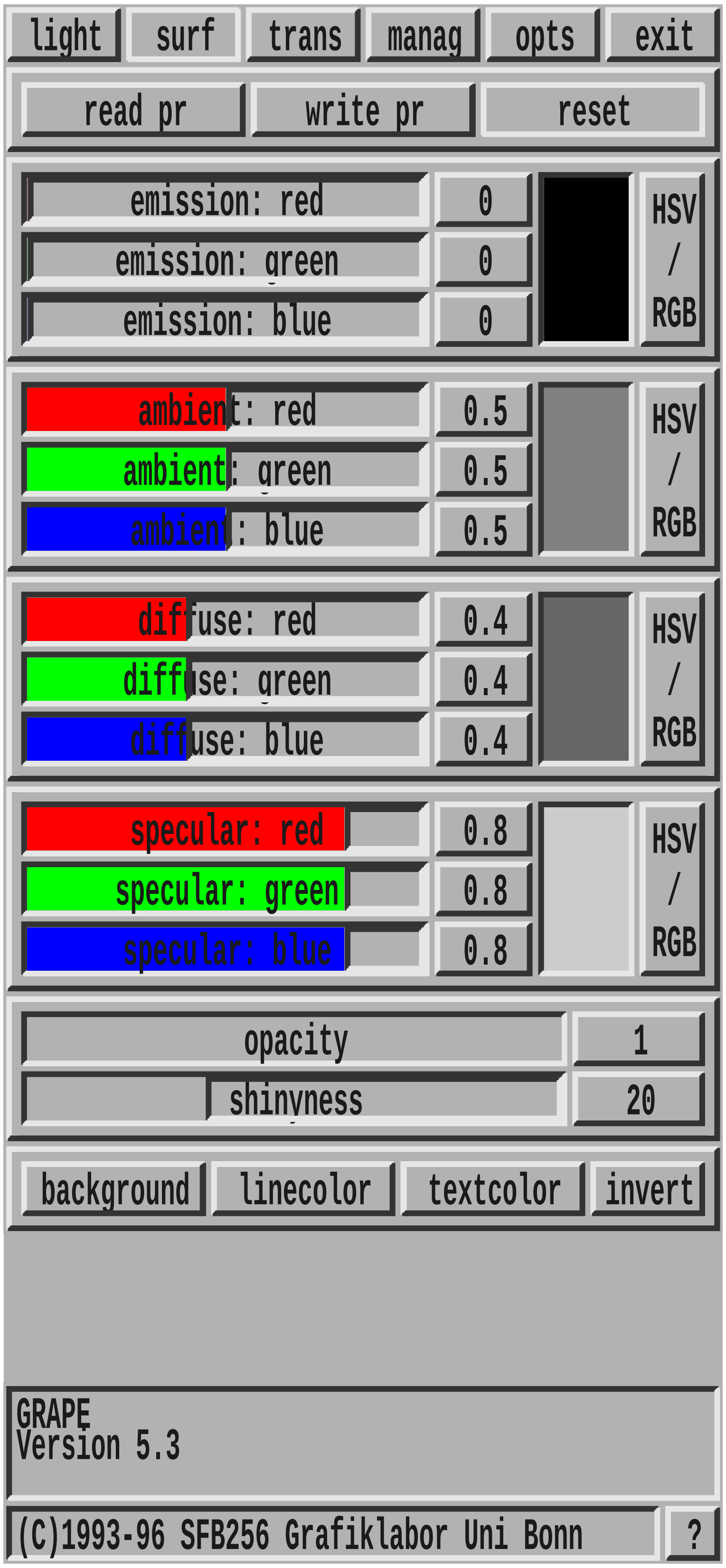


Figure 4.16: Surface Menu

4.3.8 The "Trans" Menu

The menu which controls global transformation. In GRAPE as in computer graphics in general three-dimensional objects are represented on a two-dimensional display surface. In order to get a desired view, GRAPE provides several capabilities for defining transformations of three-dimensional objects. The choice of such transformations is interactively controlled in the Trans-menu.

There are two approaches to obtaining a desired view of an object. One is to transform the object; the other is to change the position of the viewer (camera) and to choose new projections. Thats why the Trans-menu consists of two menus. The standard trans-menu allows the user to translate and rotate the object. The camera menu is based on the second approach.

`object`, `camera` This button is used to switch between the object transformation menu and the camera-menu.

`use` Switch the camera on and off. If the camera is switched off, othogonal projection is taken. The vector that specifies the look-direction is the z-axis.

`default coord-system enable` Switch the global coord-system on and off.

`config` Open a configuration layer for the global coord-system. This layer is the same as the one described in the project *coordsys*. 8.3.5

4.3.8.1 Object Transformation

The method of transforming the object is supported by the trans ruler menu and the plane or sphere rot menus.

4.3.8.2 Translation

`trans x`

`trans y` `trans z` These rulers move the object in the desired direction.

`scale` Scales the size of the object. The `N` in the scale ruler is used to make a nonuniform scaling. This scaling will be applied before the object is rotated. So this nonuniform scaling is a real object scaling.

4.3.8.3 Rotation

The user has the choice between three methods to describe a rotation of the object.

`ruler` some rulers appear (default)

`rot x` `rot y` `rot z` These FunctionRulers rotate the object around the desired axes.

`plane` This button replaces the standard rotation-ruler with a plane to determine the desired rotation.

`sphere` The Sphere-Rotation-Mode offers you a very sophisticated way to rotate the object.

`reset` Reinitialize the values of the described transformation parameters.

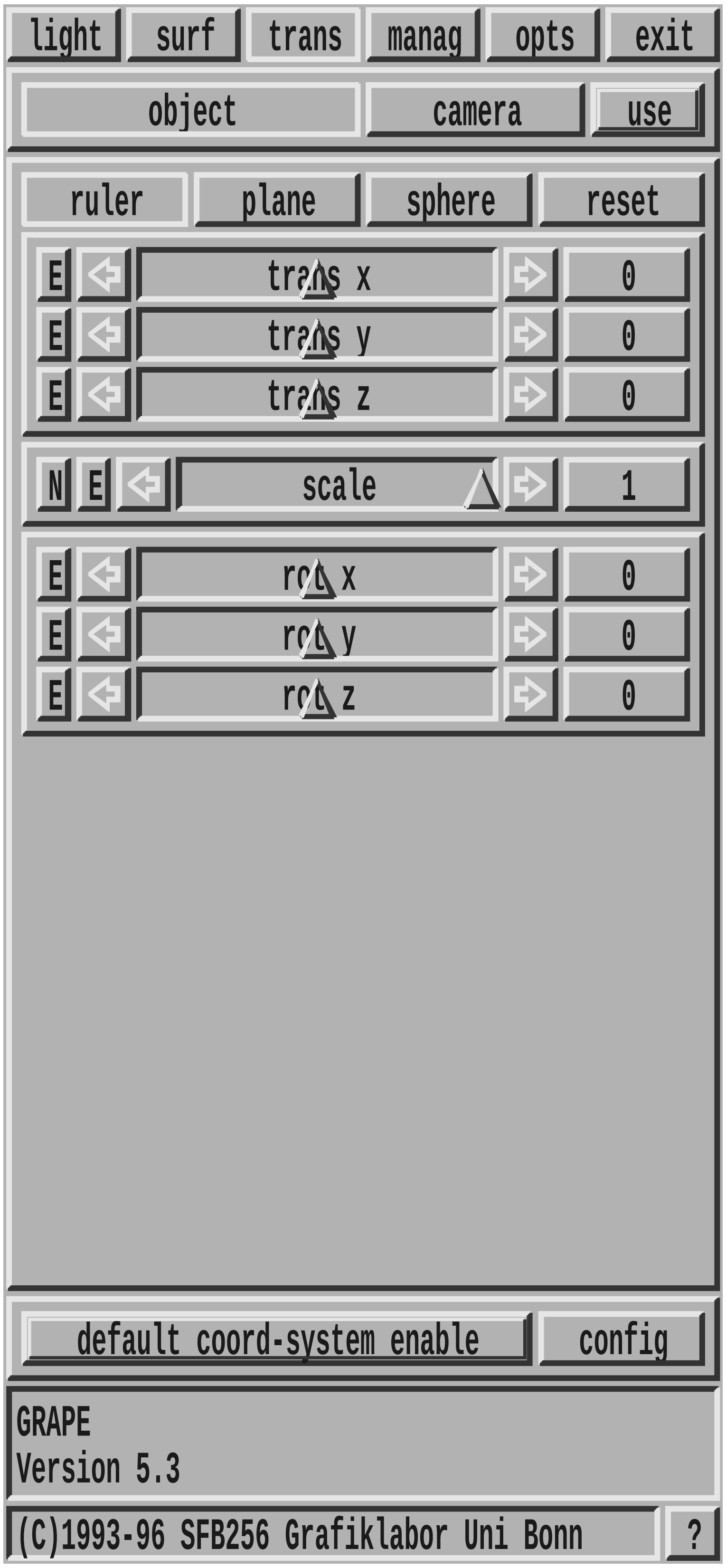


Figure 4.17: Trans Menu

4.3.8.4 How to use the "camera"

The perspective projection is determined by the following parameters: the position of the camera (eyepoint), its direction (the vector from the eyepoint to the lookpoint), a vector that specifies the direction that is 'up' in the projection plane (up-vector) and the focal length. The focal length controls the perpective in the same way a real camera does: Leaving the distance between eye and look-point stationary, a smaller focal length leads to a view, that shows more of the object at a higher grade of perspective distortion.

This menu's main part is a modified transformation menu (4.3.8).

`up` — FunctionRuler to choose the up-vector. As mentioned above the up-vector determines the "up" direction. In other words: Changing the up-vector is a kind of rotating the camera around the axis which is specified by look- and eyepoint; as a consequence the ruler controls the rotation angle.

`focal` — Ruler to select the focal length. As said above, small focal length simulates a wide-angle lens; great focal length simulates a telephoto lens.

`camera-position` — The Group below controls the position and rotation of the camera in space.

`scale distance` — Ruler to scale the distance between eyepoint and lookpoint. Remember that in camera-rot the eye-point rotates around the lookpoint in the fix distance of 1.

`look-point` — The Group below controls the position of the look-point in space.

`reset` — Reset the camera to default values. This means:
up-vector = (0,1,0),
lookpoint = (0,0,0),
eyepoint = (0,0,2),
distance = 2 and
focal = 2.

The view determined by these values is similar to the view without using the camera, except that there is perspective projection instead of orthogonal projection.

4.3.9 The "Options" Menus

This is the area where a user defined environment is built up. Six additional submenus can be activated from here by clicking on one of the buttons

that appear in the second row. These menus are dynamically defined and adjusted to the user's requirements. In most cases you will need only a few of these menus, starting with `opt 0`.

Projects include additional buttons *and* rulers here, and some display methods will install *rulers* in these menus, too. The projects you load will often install buttons in `opt 0` and rulers in `opt 1` and the other option menus; some other projects always use the same menu.

If the space seems insufficient you can try to resize the control window, the items will come back.

The system projects are described in section 8.

4.3.10 ResourceDevice and `Manager`

The `Manager` uses the `ResourceDev` (refer 9.3) to define helpful things like keyboard macros. A keybord macro can be invoked in the **GRAPE** Control WIndow by simply pressing the button,

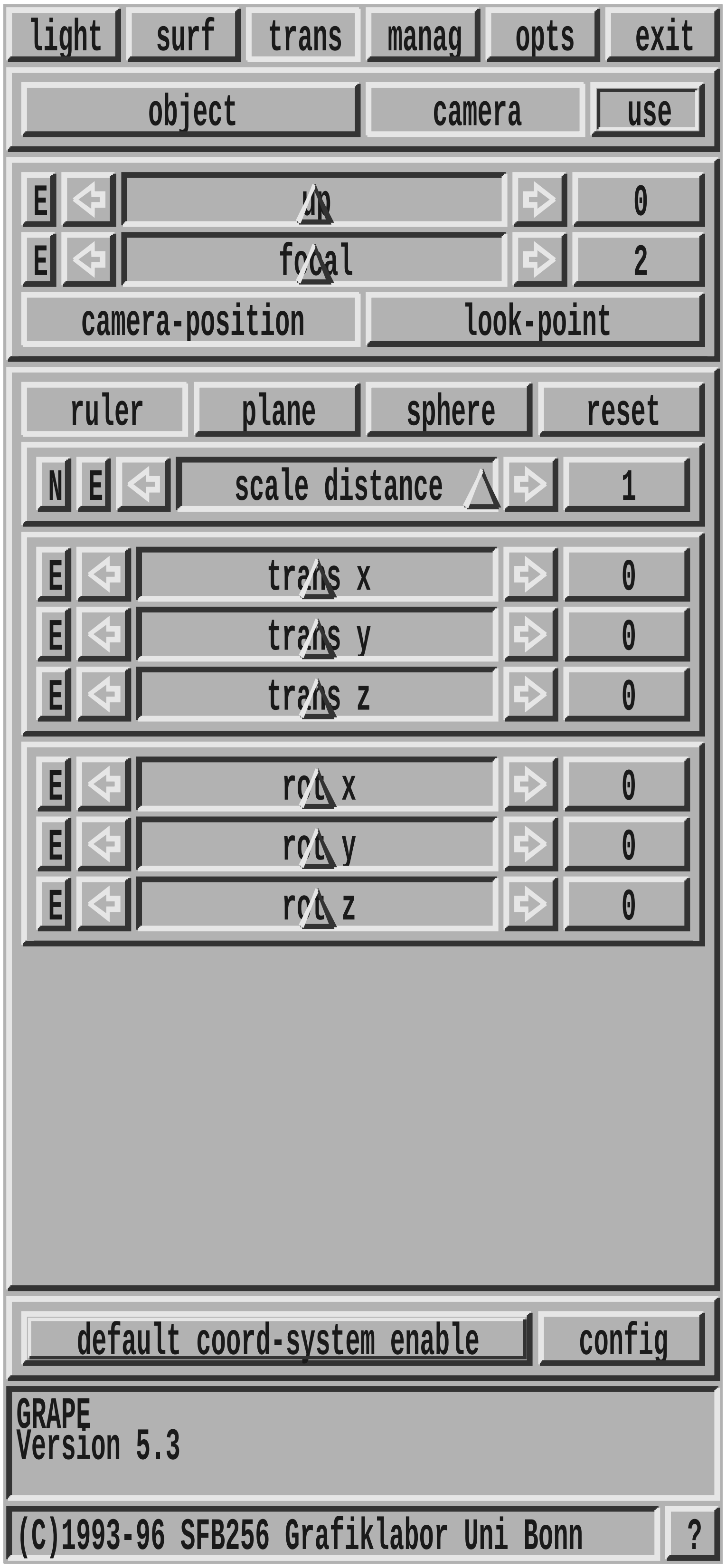
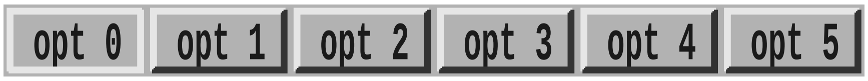


Figure 4.18: Camera



the macro is attached to. The general structure of the `ResourceDev` files is described in the section 9.3.1. The info block for the `Manager` in these files has the name Manager and has to look like that:

```
Manager = {
  tag;
   .
   .
  tag = value;
}
```

After the different tags have been described a small Manager info block to understand the stucture is given. Put it to your `.graperc` file and test it. A larger demo file can be found in our demo directory.

`help`
void
Print all available tags for the manager, their type and current settings.

`ask_on_exit;`
char
Open a confirm layer for exiting **GRAPE**. This is useful if you fear to exist your **GRAPE**–program by accident. If the tag is given without an value confirmation will be asked before exiting **GRAPE**. The default value is not to ask before exiting GRAPE.

`hotkey`
subgroup { ... }
Define a hotkey, which is a macro bound to a keystroke. Hotkeys which are bound to simple characters can not be used, if any Input field is active. The subgroup, defining the macro has the following tags:

- **`help;`**
  void
  Print all available tags which can be used in the hotkey definition.

- **`key`**
  string
  This tag must be the first entry in each `hotkey`-Group. The format of the string is `[modifier-]*key`, valid modifiers are "CASLMR". The meanings of these different modifierers are control, alternate, shift and the different mouse buttons. e.g. "C-a" means macro be executed by pressing control "a", "A-S-h" = "A-H" means macro be executed by pressing alternate and capital "h", "R-f" means the macro will be executed by pressing "f" while the right mouse button is pressed.

- **`autostart`**
  void
  Execute macro automatically at manager startup. Macros including this tag

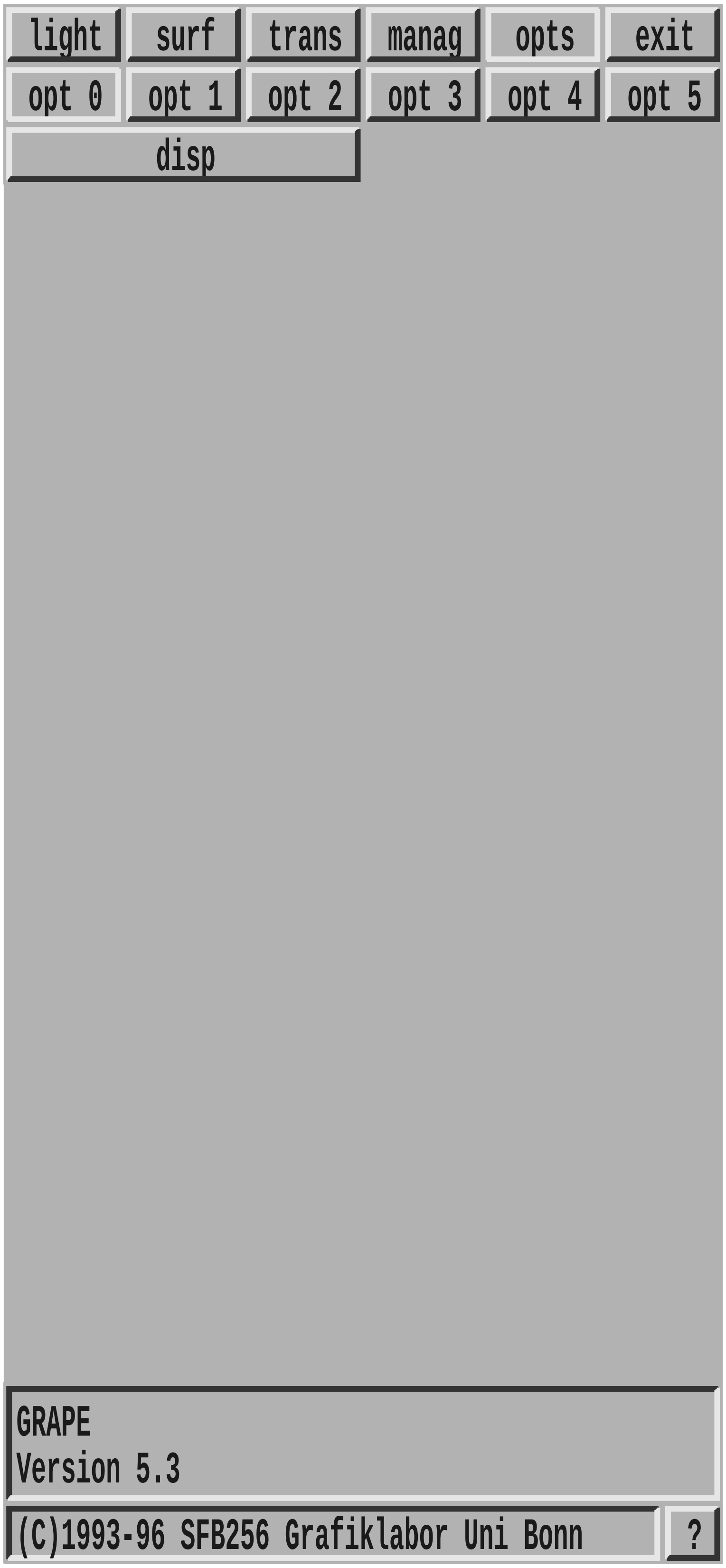


Figure 4.19: Option Menu

	can by used to configure the start up status of the manager. e.g. Switch all matrix editors to sphere mode.
`item`	string Only execute macro if item-path of item, which is currently at the mouse pointer position, matches the given item path string.
`click`	string Simulate a mouse click into the center of the given Item. Use `keystate` to select mouse buttons, the default button is "L". This tag has the same meaning as a `pressMouse` tag immediately followd by a `releaseMouse` tag for the same item.
`pressMouse`	string Simulate a mouse press event in the center of the given Item.
`releaseMouse`	string Simulate a mouse release event in the center of the given Item.
`typein`	string Simulate keyboard typein. This tag should be used to enter value into selected inputs. Pay attention, that an input is selected at the moment this part of a macro is executed. Otherwise the keystrokes will recursively call other macros if present and bound to keys existing in the given string.
`pressXxx`	int Press one of the following special keys once or the given number of times. If no value is given the key will be pressed once. Replace the `Xxx` by one of the following text {`"Return", "Escape", "Backspace", "Tab", "Delete", "Home", "LeftArrow", "UpArrow", "RightArrow", "DownArrow", "PageUp", "PageDown", "End", "Begin", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10", "F11", "F12"`}. e.g. `pressReturn` finishes the editing of an input field.
`item_send`	string Send a `"-send"`-method to the item at current mouse position. The text "-send" will by append to the method name. e.g. A keyboard macro including `item_send = "print-item-path"`

		will print out the path of the item at the current mouse position to the standard output.
`keystate`	string	Set the keystate for the following events in the current macro. Valid stateflags are "CASLMR", with the meaning described in `key`. This is useful to create special inputs like control "A".
`mousestate`	char	Set the active mouse buttons used for `click`, `pressMouse` and `releaseMouse`. Valid values are "L", "M" anr "R". The default value is "L".
`cmd_delay`	int	Set delay after each command in milliseconds. Thsi tag is very useful for software demonstrations. Using this and the following delay functions a demonstration can be tested and than shown without the danger of pressing the wrong button. Default value is 0.
`char_delay`	int	Set delay after each simulated keystroke in milliseconds. Together with the `click_delay` and the `mouse_delay` tag this feature allows everyone viewing the macro execution to understand the proceeding. Default value is 0.
`click_delay`	int	Set delay for each simulated mouse click in milliseconds. If this delay is unequal to zero you see clicked Buttons go down and after the delay time go up again. The default value is 0.
`mouse_delay`	int	Set simulated mouse moving speed in milliseconds. Increase this value if you want the mouse to slow down for the current macro execution. Default value is 0.
`pause`	int	Confirm continuation for each of the given number of steps. If the tag is given without an value 1 is assumed. Use this tag for debugging your macros or to stop demonstrations at specific points to make some explanations.

	`no_abort`	int Ignore errors for the given number of steps. Using this tag the macro will not be aborted if an error occurred. e.g. a macro wants to press a button, which may not be available. Instead of writing two different macros for similar situations put a `no_abort = 1` tag in front of the critial order.

```
Manager = {
# First a macro to get the item pathes. To use it position the mouse over
# aspecific item an press Control + "P". Remember capital "p" !
  hotkey = {
    key = "C-P";
    item_send = "print-item-path";
  }
  hotkey = {
    key = "A-x";
    mouse_delay = 20;
    click = "/main/top-row/exit";
  }
  hotkey = {
    key = "A-a"
    click = "/main/manag/misc/add-project";
    typein = "amandus";
    pressReturn;
 }
}
```

4.3.11 Output Facilities

4.3.11.1 The PostScript Driver

The GRAPE PostScript driver is capable of black&white or color pictures, intersecting patches, different paper sizes, etc. The default values may not work with your special problem, so you have to configure it for your needs. E. g. geometries with many patches but without intersections take a lot of time — hours in worst case — to be written out because the driver checks for intersection lines and for intersections. If you switch off these checkings the upper mentioned process may only take a few seconds but intersecting patches look ugly.

Advanced PostScript users will find lots of options in the output file's header as well, of course the ability to switch on the drawing of patch bounds depends on patch bounds having been written.

`Optimize Sizing`	`rotate`	Decide if the image can be rotated on the paper by 90 degree to get a bigger size.
	`stretch`	Decide if the image can be isometrically stretched to the size of the paper. This may be unwanted if `.eps`-files are created.
`File`	`filename: ...`	Input field for the filename. When `Go` is pressed more than once a number is appended.

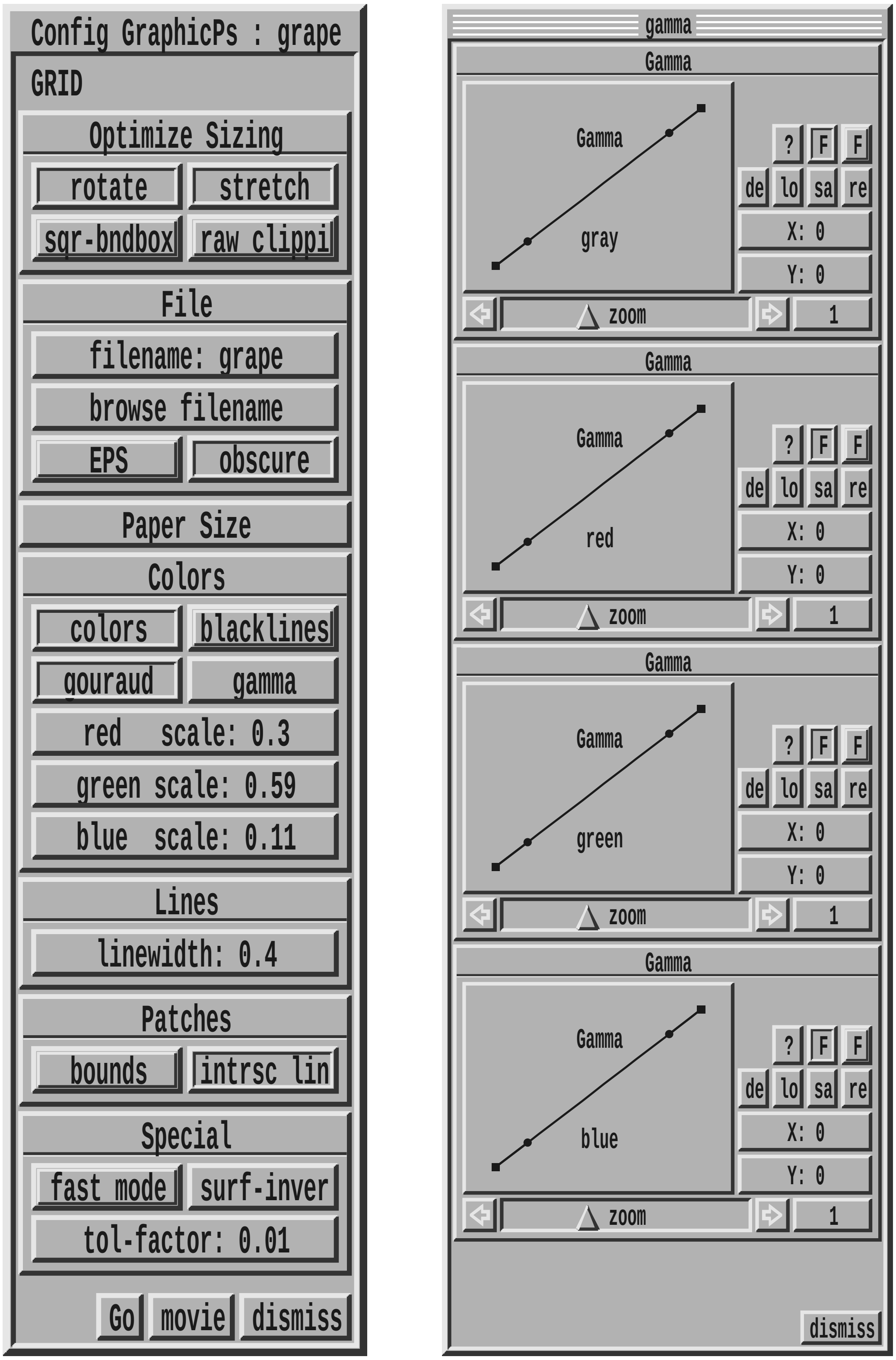


Figure 4.20: PostScript Layer and Its Gamma Curve Editor

As a difference to other filename requestors you do not have to enter the extension, the correct one `.ps` or `.eps` is automatically added.

`browse filename` — Browse through the filesystem to select a filename or a path to edit in the input `filename: ...`.

`EPS` — This Checkbox decides if the output files are marked as `.eps`-file.

`Paper Size` — Open a layer to edit the size of the paper in 1/72 inches.

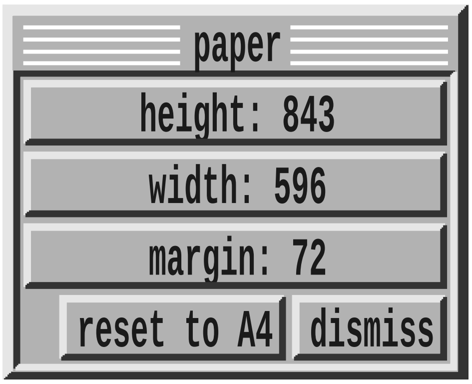


Figure 4.21: Paper Size Layer

`height` — Height.

`width` — Width.

`margin` — Size of the border on al sides of the paper. The image will be centered within the left over space.

`reset to A4` — Reset the values to A4 with border 1 inch (2.51 cm).

`dismiss` — Close the layer.

`Colors`

`colors` — Color or black and white output?

`invert` — Invert all colors?

`black grid` — If GraphicDevice is in grid-mode and `Colors` is off: draw everything black.

`gamma` — Open a layer with spline-editors for gamma-curves. In black and white mode only the grey spline is used, otherwise the three color splines.

`red scale: ...` — In black and white mode: weight factor for red color components.

`green scale: ...` — In black and white mode: weight factor for green color components.

`blue scale: ...` — In black and white mode: weight factor for blue color components.

`Lines`	`linewidth: ...`	Input the linewidth in points ($1/72 inch$) here.
`Patches`	`bounds`	Draw the bounds of patches. That is: draw grid and patch at one time. Sometimes the grid part will have holes, there is no solution for this problem except two scenes, the grid one being a bit before the patch one.
	`intrsc lin`	Calculate and draw an intersection line if two patches intersect.
`Special`	`fast mode`	Switch off intersection test and intersection lines. If they aren't needed this can speed up the output by an enormous degree.
	`specular: ...`	Correction factor for shinyness value from the surface properties. If the highlight's size is different from the real one try to adjust this number.
`Go`	Write the PostScript file. You can create numbered sequences of PostScript-files if you `Go` more then once without closing the layer between.	
`dismiss`	Close the PostScript driver configuration editing layer and its sublayers. The values are stored in memory but never written to a file.	

For OOP purposes the device writes some useless state information at the top of the layer, just ignore it.

The configuration for the `GraphicPs` can be set using the `ResourceDev`. The resource entry in the resource files totally replaces the old `ps.opt` files. The info block for the `GraphicPs` simply looks like

```
GraphicPs = {
 tag = value;
  .
  .
 tag;
  .
  .
}
```

The possible tags for the `GraphicPs` are:

`help` Print all available tags, their type and current settings.

`use_colors` int
Create a colored or a gray scale postscript file. If the tag is given without a value color mode will be enabled, default value is 0.

`patch_bounds` int
Show the bounds of the patches additionally as lines. This is useful to show the single patches of an solid model. If the tag is given without a value patch bounds will be enabled, default value is 1.

optimal_rotate
: int
Rotate the picture to get a maximum sized picture for the given paper size. Useful together with the *optimal_stretch* tag. Tag alone will enable the optimal rotation, default value is 1.

optimal_stretch
: int
Scale the picture to get a maximum picture size. Also enabled giving only the tag without a value, default value is 1.

paper_height

paper_width

paper_margin
: int
Paper size and margins in 1/72 inches. Default values are height 843, width 596 and margins 72. These are the values for a DIN A4 paper in portrait format. If the *optimal_rotation* is enabled. The landscape or portrait values can be used equivalent.

draw_cutting_lines
: int
Draw the lines, where patches are cutted from other patches. This is only useful and necessary if your geometries are self intersecting or intersected one by another. If the *cutting_check* is disabled, the value will be ignored. Default value is 1.

eps
: int
Create an Encapsulated PostScript file instead of a normal one. If only the tag is given eps will be enabled, default value is 0.

specular_correction
: double
Correction for the shinyness value. This option is used, if size of the highlight on your surface differs from the one in your graphic window. Default value is 1.0.

linewidth
: double
Configure the linewidth of the lines. Default value is 0.400000.

invert_colors
: int
Invert the colors. This is useful if you don't want to adapt your surface properties and line color, but simply want to create a picture from your graphic on a white background. If the tag is given without a value, colors will be inverted, default value is 0.

black_lines
: int
If the grid mode is configured and this flag is enabled all lines will be drawn in black, even if they were colored or gray. Tag alone means enable black lines, default value is 1.

huge_patch_outline_mode
: int
default value 0

no_cutting_check
: int
Do not make an intersection check. This can speed up your output time dramatically, but can only be used, if you know, that your surfaces do not intersect. Also intersections between two different object are not resolved. Intersection check is also disabled, giving the tag without a value. Default value is 0.

gouraud int
Enable the PostScript gourand shading. This will result in very smooth and pretty pictures, but will increase the printing time. This shading is done inside the printer, so use it only if your printer is rather fast. This shading is not visible using a ghostview, but could be seen using a ghostscript to render it into a bitmap and view this bitmap. If the tag is given without a value gourand shading will be enabled, default value is 0.

obscure int
If this flag is enabled, patches, which are obsscured will removed in front of creating the file. This will encrease the time used to create the postscriptfile, but may improve the printout time on the printer. Use this option if the file should be printed in a very high solution or if your printer is much slower than computer. Be aware, that the PostScript file is created once, but may be preinted a several times. Only the tag given will be interpreted as obscure check enable, default value is 0.

square_bndbox int
Using this flag a file with a squared bounding box is created. This bounding represent exactl the border of the graphic window. Using this option the graphic will have the same margin as in the graphic window. It is very usefull if you want to print several different graphic output together in the same page. If the tag is given without a value the squared bounding box will be enabled, default value is 0.

raw_clipping int
Normally the patches are cutted at the clipping box. Use this flag to remove all patches which have at least one vertex outside the clipping box. With that option an inside view of an object through a cutting hole will look much more like a technical picture. Also with this tag the tag alone enables the option and the default value is 0.

depthsort_tolerance_factor double
Correction factor for the depthsorting algorithm. Use it if you have problems with the depthsorting of patches. Teh default value is 1.0.

rgb_to_gray VEC3
The weight factors for the RGB to gray calculation. Only used, if the color output is disabled. Default values are: 0.30, 0.59, 0.11.

filename string
Filename for the Postscript file. A .ps or an .eps will be appended depending on the Encapsulate mode off or on. The default name is *grape*.

4.3.11.2 Videos and Bitmaps

The functions for creating videos or bitmap files of arbitrary size are collected in sometimes very platform specific projects, like `video` (video functions, using an IRIX utility), `rgbvideo` (create arbitrary sized bitmaps or bitmap file sequence videos on (for now) GL graphic) or `LVR` (locally written for one special laser video recorder).

Look at the projects listed in section 8.

4.4 The Concept of Time

4.4.1 What Does "time" Mean?

This question may seem a little bit curious to some people, especially to scientists studying physical dynamic phenomena. Such problems carry in general an inherent time, just the physical time of nature. Also a video animator usually has this notion of *time* in his mind when designing a video. This is manifested for example in the unique time slider in many software animation packages.

In these example the term *time* is always used as the physical time, or at most to some extent modified, which nature gives us. We extend this characterization of time and define: *time* is an arbitrary but emphasized parameter of the object we study.

As the first consequence of the extended meaning of time let us assume our object depends on a set of parameters. We can emphasize one of those and consider the object as a dynamic process depending on this parameter, now called the time–parameter. During interaction we may choose a different parameter or the other parameters may for instance also depend on the time–parameter.

A second characteristic of our time–parameter is *locality*. In the extreme for example, every object belonging to a hierarchy may have an own time–parameter. This may seem unnecessary when creating a video which relies on a global time. But when analyzing dynamic processes it is important to see for example the same process running at different speeds or times simultaneously, or fix the time of one of the processes while keeping the others running.

Our definition of a time parameter extends the term *time* from the physical time to include any real parameters. This arbitrariness for instance allows in multi–parameter systems changing between parameters to analyze different aspects of the problem with the same tools. Surely, the new freedom in the definition of *time* shall not make things more complicated and possibly only introduce a new parameter. When studying flow problems one would still choose the physical time as *time*. But even here one might want to see two copies of the flow evolving simultaneously with different times, i.e. speeds.

Summarizing, we use *time* as follows:

- *time* is a word for an arbitrary emphasized parameter of a parameter–dependent object
- `time` is a parameter belonging locally to an animated object.

4.4.2 Classes Managing Dynamic Processes

There are different classes implemented in **GRAPE**which supervise and manage dynamic processes. Instances of these classes will appear as nodes in the data hierarchy.

4.4.2.1 TimeScene

The class `TimeScene` is a subclass of the class Scene with additional instance variables: a dynamic object, a local time and a synchronization flag. The dynamic process can be arbitrary, an implementation of such a process is described below.

4.4.2.2 TimeObject

It contrast to `TimeScene` `TimeObject` expects the dynamic process to consist of a fixed number of discrete steps (keyframes), the geometry may still be arbitrary. TimeObjects are no Scenes, this class is implemented differently but can nevertheless be handled in almost the same way as `TimeScene`.

4.4.2.3 Surface

`Surface` is a subclass of `TimeScene`. For Surfaces the discrete steps must have a specific format — a linear list of TimeSteps with `Geom2d` geometries. Classes that are based on `Surface` are mainly used for computing and visualizing surfaces.

4.4.3 Creating Classes That Implement Dynamic Processes

Creating a class that implements a dynamic process means to implement a class which provides the method `"get-object"`(time). This method returns the object for the given time from the dynamic process. This is all you need, if you have written `"get-object"` you may use this new class as a dynamic object for `TimeScene`. As mentioned the classes `TimeObject` and `Surface` require a certain type of data structure as dynamic process, therefore they can not be used in this generality.

4.4.4 The Class TimeStep Extending Key–Frame Techniques

The dynamic process `TimeStep` consists of a doubly–linked list, each node having additionally to the list pointers an instance variable time and a pointer to an arbitrary object. The value of time is obviously the time of the object at that key–frame. Linear interpolation between two key–frames is therefore done by simply interpolating the corresponding objects. With this information we can describe the action of the method "get object"(time) being send to the dynamic process `TimeStep`: if the given time is equal to one of the key–frame times the corresponding key–frame object is returned. Otherwise time is between two different key–frame times and the interpolation algorithm has to be applied to generate a time–cut. Most of the geometry classes support the interpolation of different objects with the same discretization. Especially on the triangulation classes these techniques are implemented, the linear interpolation may easily be extended to higher order techniques.

The concept of `TimeStep` allows to include classical key–frame techniques in our time–concept in a natural way. As before, we only require the method `"get-object"`(time) to exist on `TimeStep`, i.e. the key–frame objects need to have some kind of interpolation mechanism.

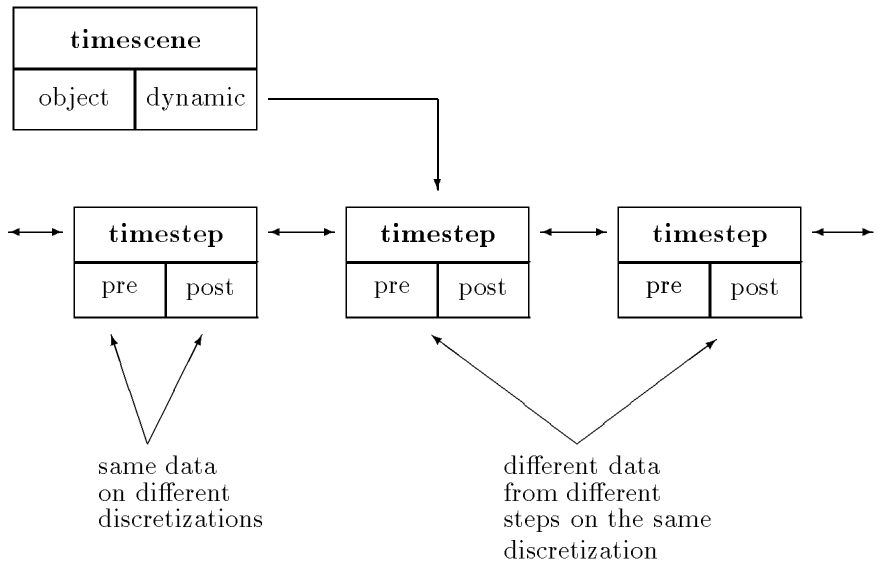


Figure 4.22: Dynamic Instance of a TimeScene

4.4.5 Adaptive Strategy for Timedependent PDE

Despite the fact that we store of course also key frames, the behaviour of an instance of `TimeStep` is that of a dynamic process: methods send to such an instance act on the instance as a whole, only the instance knows what kind of activities will be released after having receipt a message.

We will further on examine the underlying data structure for a significant application. Therefore let us briefly inspect a large class of applications from the calculus in continuum mechanics. Our intention is to focus on the numerical solving of timedependent partial differential equations such as parabolic or hyperbolic equations and systems, especially flow problems. This will lead us to the appropriate structure for a single timestep of such a process. In a more abstract setting the latter equations turn out to be the ordinary differential equations in function spaces

$$\frac{\partial}{\partial t}u(t) = A(u(t),t)$$

were in addition u fulfills appropriate initial and boundary conditions. Here A is a nonlinear operator depending on the specific problem. We will assume that an adaptive strategy in space and time is used. Then a list of instance of the class `TimeStep` is the appropriate choice for the instance variable `dynamic` in the `TimeScene`. A `TimeStep` contains its `time`, a `pre_object` and `post_object` (in the figure below shortened to pre and post), furthermore a binding to the previous and the next timesteps called `pre_step`, `post_step`.

Let us consider as an ODE solver for example a general adaptive One Step Scheme to calculate the numerical solution $u(t)$ on a discrete in general unstructured grid $G(t)$. One step starts at time t with $(u(t), G(t))$ and calculates at the time $t + \Delta t$ a next timestep $(u(t + \Delta t), G(t + \Delta t))$. Involving a standard nonadaptive One Step Scheme and starting with u_{post}, G_{post} at time t a frequently used algorithm for a single step looks as follows

$$\begin{aligned}
\Delta t &= \text{TimeStepEstimate}(u_{post}, G_{post}, t) \\
t &= t + \Delta t \\
u_{pre} &= \text{OneStep}(t, \Delta t, u_{post}, G_{post}) \\
G_{pre} &= G_{post} \\
G_{post} &= \text{AdaptMesh}(u_{pre}, G_{pre}) \\
u_{post} &= \text{Projection}(u_{pre}, G_{pre}, G_{post})
\end{aligned}$$

where **TimeStepEstimate** returns the new possible step length, **AdaptMesh** does the refining, coarsening or deforming of the current mesh according to heuristic criteria or a–posteriori error estimates and **Projection** projects the result of the **OneStep** onto the function space on a different mesh. In our setting for example a function $u(t)$ and the underlying geometry may be condensed in one `Fe2d` or `Fe3d` object `fe(t)`. At the current stage the underlying geometries should be of simplicial type. But the concept is not restricted to them and more general types of geometries are in progress. Therefore at any timestep we have two such objects $fe_{pre} = (u_{pre}, G_{pre})$ and $fe_{post} = (u_{post}, G_{post})$ which were included in the current `TimeStep` structure with time t as its `pre_object` and `post_object`.

The method `"get-object"` on a list of timesteps with underlying `Fe2d` or `Fe3d` objects then again returns an object of the same class. Because of the same discretization of the two objects `pre_object` at time $t + \Delta t$ and `post_object` at time t a standard and general interpolation is now defined. Not differing between `pre` and `post` interpolation is in general senseless to define. The discussion above has shown that the necessary structure is naturally given in most cases.

Finally all the static visualization techniques such as level set, isoline and vector field rendering or extracting approaches to display the discretization once implemented on this class now immediately work on time cuts of the latter processes. A `TimeScene` containing as its `dynamic` instance a list of `TimeSteps` can be sketched in figure 4.22.

4.4.6 Time Dependent Interactives

The simplest time dependent interactive objects are animated variables (integers, floats or doubles). This class is a bit exceptional, since displaying a variable has no meaning in a graphic scenery, so this class should of course not be used inside the scene hierarchy. It is described here, because it has the same philosophy as the previous classes.

A time dependent variable is controlled by a function. Controlled means, that the function is in some sense hidden for the user. The user only works with a variable, but the variable will be updated by the system automatically when time changes.

Animated variables are implemented in **GRAPE**by the class Function_Ruler. The function is a spline, it can be modified by a spline–editor. To activate the automatism the user only has to create the function–ruler which controls his variable.

4.5 Projects

One of the fundamental concepts in **GRAPE** is its project management. In this section we will explain what projects are, what functionality they provide, and how they can be used. We give a short overview of the predefined system projects, this should help to quickly find out which projects might be used for solving a specific problem. The full documentation of all projects can be found in section 8.

More information about the internal project mechanisms and details about writing projects can be found in section 5.6 in the chapter "Programming with **GRAPE**".

4.5.1 What is a Project?

Imagine a user has written some methods for creating and editing starting triangulations for her/his numerics. Some buttons are needed to invoke these methods and some flags and rulers to control what the methods do when they are called, additionally a basic triangulation to start with might be useful. All this could be added to the **GRAPE** program in the main routine, but then the triangulation and buttons and rulers are always present, they fill up the menus even if the user is working on something else and they are not needed. A separate main program could be created for each application, but if two packages are needed at the same time a new main program must be created with a copy of both codes. This gets more complicated with each application that is added, and if packages are exchanged between several users it gets unmanageable.

The solution for these problems are projects. A project is a collection of classes, methods and interactive elements (a "user-interface"). Projects can interactively be added to and removed from the system, thereby adding resp. removing the classes and methods and modifying the interactive environment. A project may depend upon other projects which are loaded resp. unloaded automatically with the project. New objects can be inserted into the data hierarchy when a project is added.

4.5.2 Advantages of the Concept

Using projects helps to organize **GRAPE** and to enhance its modularity. Combining all classes, methods and the interactive elements to a project forces an application to be self-contained and to have a well defined interface.

Projects can easily be exchanged between users. The only thing a user has to do to add a project to an executable is to link the files of the project (or even better: a library with these files) to the executable and to declare the new project with a single line of code. Then the new project is available with a mouse click — it even can be added automatically at startup —, and there is no need to change the personal working environment.

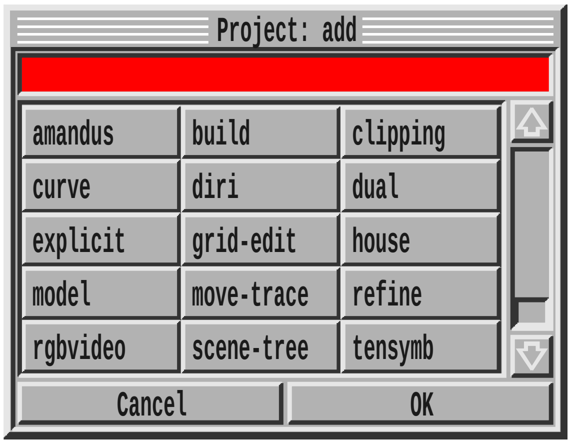
For a user who writes a new application we therefore strongly recommend to create a project for her/his work. How this is done is explained in detail in section 5.6.

Currently we are working on a DSO manager which allows to load, remove or replace projects which are linked as dynamic shared objects during runtime. This will further improve the modularity of **GRAPE** and simplify exchanging projects.

4.5.3 Using a Project

There are two buttons in the main menu of **GRAPE**'s manager which are used to add and to remove projects:

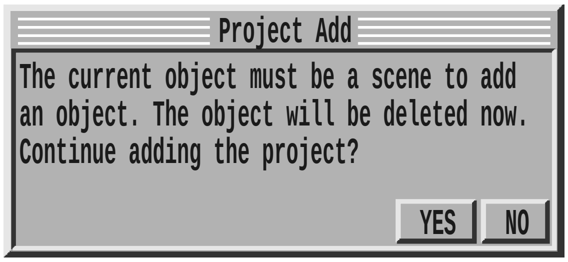
When the `add project` button is pressed a list of all available projects appears. It may differ from program to program depending on which projects were compiled into the **GRAPE** library or added locally, but it should look somewhat like this:



Now you can select a project to add it to the system (the same project can be added several times though this only makes sense for projects which add objects to the data hierarchy tree). Removing projects is as simple as adding them: press the `del project` button and a list of all added projects appears, then select the one you want to delete. If a project was added more than once it has to be removed as often as it was added to really delete it from the system.

Projects can be declared as hidden, then they do not appear in the add list. In most cases this is done because they only make sense when they are used by other projects or because they are still under development and not intended for general use. Nevertheless they can be added by typing in the name, try for example *surface* (this project provides the user–interface for the abstract class `Surface`).

In addition to classes, methods and interactive elements a project may also provide its own objects, to add them to the data hierarchy tree the current object has to be a scene. Otherwise an error requester appears which offers to either add the project without its objects (that is its



classes, methods and interactive elements) or to abort adding it. The objects to add will be deleted in both cases.

Some projects install a user–interface in one (or several) of the option menus, where you can find it and how the project interface is used is described in the project's documentation in section 8.

4.5.4 Overview of Existing Projects

Now we will give a short overview of the predefined projects. Some of them might be missing in some **GRAPE** libraries, for examples those depending on specific hardware like *RGBVideo*. The following list is sorted alphabetically, for each project the reference to its documentation in section 8 is given.

Amandus (8.2.6): Compute parameter–dependent minimal surfaces from their Weierstrass representation. The parameter and domain information are read from a description file and can be edited interactively.

Build (8.2.4): Surface builder for creating parameter–dependent triangulations from macro patches (triangles and quadrilaterals at the moment). The patch data is read from a description file and can be edited interactively.

Clipping (no documentation)

Curve (8.2.2): Compute parameter–dependent curves from explicitly given formulas read in from a description file.

Diri (8.2.9): Combines *Build*, *Refine* and *Dual* to an environment for computing discrete minimal surfaces and their conjugates.

Dual (8.2.7): The Dirichlet minimizer for computing discrete minimal surfaces and their conjugates. Starting triangulations can for example be created with *Build* or *Explicit*.

Explicit (8.2.5): Compute parameter–dependent surfaces given by explicit formulas. The formulas, parameter and information about the domain are read from a description file and can be edited interactively.

Grid_Edit (8.1.2.2)

House (no documentation): Demo project, see section 4.3.3.

Images (8.3.2.1): Internal project used by *RGBVideo*, it implements the file handling.

Model (8.3.1): "Hard" transformation of geometries. In contrast to the surface menu or the scene transformations the object's point coordinates are transformed.

Move_Trace (8.1.1)

Probe (8.1.3)

Refine (8.2.8): Interactive refinement, included are methods to refine regions of high Gaussian curvature, the boundary or the area inside a ball.

RGBVideo (8.3.2): Project for creating bitmap files of arbitrary size and video sequences. Only available on IRIS GL capable machines at the moment.

Scene_Tree (8.3.4): Data hierarchy viewer, very useful when working with large hierarchies, especially for **GRAPE** novices.

Surface (8.2.3): Basic project for working with parameter–dependent surfaces. Several geometries can be handled at the same time, the parameter are read from a description file.

Time_Object (8.2.1): Basic project for working with parameter–dependent geometries. The parameter are read from a description file.

Tr2d_Edit (8.1.2.1)

Chapter 5

Programming with Grape

This part of the manual is dedicated to programming with GRAPE. First a general introduction to object-oriented programming is given, then the implementation of the basic concepts in GRAPE is presented. The implementation of new classes and methods, handling interactive elements and creating projects will be explained in detail.

You should be familiar with C programming, this is all you need to start programming with GRAPE. You will find many hints concerning notation, naming convertions, available methods and functions etc. in this chapter, we hope that this will help to get over difficulties.

A complete example program for computing a surface of revolution from a curve is developed, starting with the implementation of the new class, the basic methods, interactive elements to edit points, finally creating a new project our of this example.

5.1 Object-Oriented Programming

5.1.1 Sequential Versus Object-Oriented Programming

For standard programming techniques functions are the most important parts. They contain a sequence of statements which manipulate certain data structures. The main tool for abstraction is the usage of subprograms, this allows to use the same statements at different locations and divide one task into different operations.

In contrast to sequential programming the data structures are the central elements of object-oriented programming. The data is mentioned as individual objects in a certain state. According to their structure each object belongs to exactly one class. It is referenced as an instance of this class. A number of operations (methods) are assigned to each class, which modify the state of the corresponding instances. Moreover the classes themselves are mentioned as objects, because there exist special methods, which operate on classes, e.g. create a new instance.

The aim of object-oriented programming is to transform the objects from an initial to a final state by the application of appropriate methods.

5.1.2 Classes, Instances and Methods

The classes are sorted hierarchically in a tree. Each class corresponds to one of the tree's nodes. A subclass is represented by a child's node and the parent's node is the superclass. The root of this tree is a class whose instances have only a basic structure. Every subclass is defined by its superclass attributes extended by further properties. In general this is an extended data structure with new and advanced methods.

The classes may have instances, which represent the same class data structure at different states, e.g. several graphic objects of different type. Likewise each instance belongs to one class, which defines its structure. The structure is made of variables called instance variables. These variables define exactly one state of this object. The application of methods affects only the instance variables, e.g. a method modifies a graphic object by updating the corresponding instance variable according to the parameters.

The hierarchy of the class structure is guaranteed by integrating the instance variables of the superclass. This means that an instance of a class consists of the instance variables of the superclass and the additional instance variables of the class.

5.1.3 Main Concepts of Object-Oriented Programming

The main concepts of object-oriented programming are

- **Message Passing**

  In an object-oriented programming environment the execution of any object method is implemented by passing a message, which contains the method name and the necessary parameters. This message invokes searching for the appropriate method in the class' list of methods. If a suitable method is found, it is executed.

- **Inheritance**

  Remembering the last paragraph about instances it is possible to apply a method to the subclasses instances, because they have at least the same variables. In fact this feature is the most important abstraction in object-oriented programming. If an object receives a message and the appropriate method is not found, the method name is searched in the list of the corresponding superclasses methods. These methods can be applied, because the hierarchy of the class structure is guaranteed. This is the way subclasses inherit their superclasses methods. Moreover the superclasses methods can be called directly.

- **Late binding**

  Since a method's application is set up by "message passing" and the correct method is chosen according to the object class during runtime another interesting feature has to be mentioned. If a class instance has a variable pointing to another object, the actual method can send a message to this object. It is not necessary to care about the object class during implementation, because the corresponding method is found automatically. This means that a method name is identified with a function during runtime (late binding). Moreover the applied function depends on the corresponding class of the object. Therefore different methods may have the same name in different classes, e.g. different methods are needed to display different objects, but they all have the same name "display".

5.2 GRAPE

GRAPE provides a programming environment to develop object oriented graphic programs. It provides data structures for classes, instances and datatypes and a large number of methods and utility functions. Most of the package is implemented in ANSI C though some old parts are still Kernighan/Ritchie style C.

5.2.1 The `GRAPE` Function

The GRAPE kernel consists of a set of functions which handle the message passing, i.e. invoke the appropriate function for the desired method on an instance or class.The main function will be invoked by

```
GRAPE(obj, message)
```

where `obj` is a pointer to the object which should receive the `message` given by a simple string. This string defines the name of the method. This `GRAPE` command supplies a pointer to the appropriate function which implements the method. The method will be executed if a parameter list (which might be empty) is supplied. Therefore a typical grape command looks like this

```
GRAPE(obj, message)(parm1, parm2, ...)
```

With an empty parameter list — don't forget the empty parenthesis — it should look like this

```
GRAPE(obj, message)()
```

The structure of the `message` string defines the precedence to search for the corresponding function. No option given starts searching in the class of `obj`. If the message has a leading `"="`, the method is searched in the actual method's class. A leading `"^"` commences search in the superclass of the actual method class, multiple `"^"`'s can be used to skip superclasses. Additionally it is possible to use `G_CAST_CLASS(obj, class)` as object, then the search starts in class `class`. This should be preferred to using multiple `"^"`'s because it is easier to read and independent from changes in the class hierarchy.

Since the list of parameters isn't limited at all in grape, for implementing a new method all sub- and superclasses should be searched for methods with the same name. The new method should have the same parameter structure as these already existing equally named methods, this helps avoiding problems with inheritance.

There are two kinds of methods that can be used interactively. So called "display methods" are used to update the graphic window (see section 4.3.2), they should be marked by the suffix `"-disp"`. The standard display method is called `"display"`, it works on nearly every existing geometric object and should be implemented for every new geometry class (see section 5.4.6). Methods that can be send interactively to an object (e.g. a numerics method that can be started by a button press) should have the suffix `"-send"`. Both types of methods obviously must be parameterless, parameters for them can for example be provided by rulers (see sections 5.4.6 and 5.5.5).

Now, what happens when you call `GRAPE(obj, message)`? Let `message` be the non-prefixed method `"name"`, `obj` an instance of class `Dummy`. Then

- GRAPE searches the methods defined on the class `Dummy` of `obj` for the method `"name"`.
- If there is no method `"name"` on `Dummy`, perhaps there is one on the superclass of `Dummy`? Or on the superclass of the superclass? And so on.
- If no method `"name"` was found `GRAPE` looks for a method `"universal"` (see section 5.4.8), again first the class and then the superclasses will be searched.
- Having terminated the search, `GRAPE` pushes `obj` on an internal stack together with information on the method, primarily the method's class, i.e. the ancestral class the method is inherited from.
- Finally, `GRAPE` returns the address of the function related to the method. If neither `"name"` nor `"universal"` were found an error message is printed and a pointer to an error handler (an empty method) is returned instead.

The function returned by `GRAPE` when called obtains the `obj` by `START_METHOD` which just looks up the internal stack. The complementary `END_METHOD` pops the last item (`obj` + method information) from the stack. The use of `START_METHOD` and `END_METHOD` is illustrated in section 5.4. As mentioned before the class the search starts in can be influenced by using the prefixes `"="` and `"^"` or `G_CAST_CLASS`.

There some points about the `GRAPE` call you should be aware of:

- Since the function which is determined by `GRAPE` is not known at compile-time, the compiler cannot do any type-checking for the argument list. Yet, the compiler performs the usual type promotions (float $\rightarrow$ double, short int $\rightarrow$ int, char $\rightarrow$ int), so keep to self-promoting types in the parameter list of your ANSI C method routine (alternatively it could be declared in the obsolete K&R style, then the same type conversions are done, but this in not recommended).

- A common error is to forget an empty argument list `()` to the `GRAPE` call. In this case the object returned by the method is the function pointer, continuing to work with this "object" has crashed one of our SGI machines before...
- It is not allowed to store the function pointer returned by `GRAPE(obj, message)` and to call the corresponding method function more than once. From the second call on `START_METHOD` returns a wrong object to the method, the program might crash if this object has the wrong class.
- Don't send prefixed method names to objects of another class than the object obtained by `START_METHOD` and only use `G_CAST_CLASS` to cast the object to one of its superclasses.

5.2.2 Memory Management in GRAPE

To avoid problems with memory fragmentation and the different memory managements on different computer systems GRAPE has got its own memory management. The basic GRAPE function for allocating memory is

```
void *mem_alloc(size_t size)
```

which allocates a memory block of size `size *sizeof(char)` and returns the pointer to the block, if the allocation fails a `NULL` pointer is returned (the syntax is identical to the system `malloc` function). To free a block allocated by `mem_alloc` you have to use

```
void mem_free(void *ptr, size_t size)
```

where `ptr` is the address and `size` the size of the block to be freed. The difference to the system `free` function is that the size of the block to be freed also has to be passed to `mem_free`. This is necessary because it is (for historic reasons) possible to free only a part of an allocated block, though this is not recommended. To free a block without knowing its size you can use the special value `G_BLOCKSIZE`, but giving a size enables `mem_free` to check it. You can change the size of a block using

```
void *mem_realloc(void *ptr, size_t old_size, size_t new_size)
```

which basically is the same as allocating a new block, copying the data, freeing the original block and returning the new one. Though `mem_realloc` tries not to move the block, shrinking will never move it. In case of an error `mem_realloc` returns `NULL` and the old value of `ptr` will still be valid.

When working with GRAPE classes and datatypes it is necessary to use this memory management instead of the system functions — of course it is not possible to mix both, freeing a block allocated with `malloc` with `mem_free` or vice versa will certainly result in an error (or crash). An advantage of the GRAPE functions to the system functions is that they provide a much better error checking: when an error occurs the type of the error and information about the last method called and the object it was called on is printed.

There are special memory functions based on these functions for most C and GRAPE datatypes, see section 10.2.1. When implementing a new datatype you should also provide the corresponding memory functions.

5.2.3 GRAPE Datatypes and Utility Functions

Additionally to some utility classes like `G_List` for arbitrary linked lists (see 10.1.1) and classes for arrays of objects like `List_Of_Inst` (see 10.1.2) there are datatypes in GRAPE which for various reasons are not implemented as classes. The most important types are two-, three- and four-dimensional vectors of integer (`INT2`, `INT3`, `INT4`), double (`VEC2`, `VEC3`, `VEC4`) and float (`FVEC2`, `FVEC3`, `FVEC4`) variables, three by three and four by four matrices (`MATRIX33` and `MATRIX44` for double and complex (`COMPLEX`) variables.

The vector types are mainly used for working with points, the matrix types for transformations. There are lots of functions which operate on these types, you should have a look at section 10.2 to see what's available. Some time critical functions are implemented as macros, but none of these will evaluate its arguments more then once.

Note that (almost) all GRAPE utility functions start with the prefix `g_` followed by the type they operate on (e.g. `g_vec3_` for the `VEC3` functions). This convention should be used whenever new functions are written or new datatypes added, it helps to avoid naming collisions.

5.2.4 GRAPE Math Functions

In section 10.2.15 you will find a list of all elementary mathematical functions in GRAPE versions, all with that prefix `g_`. These functions were added to GRAPE because the names and types of the basic math functions are not standardized. Using the GRAPE functions helps making programs portable.

It should be mentioned that wherever possible these "functions" are macros which are replaced by the appropriate system math functions at compile time, so there is no overhead added to the program when they are used.

5.3 Creating a New Class

Now that the basics of object-oriented programming and their implementation in GRAPE have been explained, we will describe what a class looks like in GRAPE and how new classes can be added to GRAPE. Defining the class `Rot2d` will be the first step in creating our demo program. Methods will be explained in the next section.

5.3.1 Design and Implementation

Designing and implementing a class are two different steps. First you have to decide what the instances of the new class should look like, that is what data should be stored in them, and which methods, that is operations on the data, are needed.

Then you have to find a suitable superclass for the new class, this superclass will be extended by adding the instance variables of the new class to its structure and by implementing new methods.

The new class inherits all the functionality of its superclass, because all existing superclass methods can be called on the new class, too. A superclass method may be overwritten by implementing a method with the same name for the new class, this method of course may call the superclass method to extend its functionality for the new class instead of replacing it. This inheritance ensures that you have to write the least amount of code necessary to make the new class work, and it will lead to more stable programs because you can rely on the already working superclass code. For these reasons choosing a superclass is probably the most important decision you have to make when designing a new class.

In case there is no suitable superclass for the new class the base class `Root` of the class hierarchy has to be used — `Root` is the superclass of all GRAPE classes.

5.3.2 Naming Conventions

Before we continue explaining the implementation of a new class some naming conventions in GRAPE should be clarified. To distinguish between a class, its instances and the typedef of the instance structure we write

`Class` — for the pointer to the class,

`class` — for an instance of Class and

`CLASS` — for the typedef of the underlying structure.

5.3.3 The Data Structure for a New Class

When the superclass for the new class has been chosen first the struct describing the instance variables has to be defined. This struct has to contain the struct of the superclass (to be more precise, of all superclasses including the base class `Root`). For every class, the Grape header defines a macro constant that contains the declarations of all instance variables:

```
#define MY_CLASS_STRUCT \
  SUPERCLASS_STRUCT;    /* INSTANCE_STRUCT if      */\
  <decl>;               /*   Root's the superclass */\
  <decl>; \
  <...>
  <decl>                /* here no ';' */


typedef struct my_class {
  MY_CLASS_STRUCT;
} MY_CLASS;
```

Since a change in the definition of a class affects all subclasses you should not paste the definition of SUPERCLASS into your new typedef but adopt this declaration style. Then it will be sufficient to recompile your code if, for example, a new version of Grape has changed the definition of Root resp. instance.

Your declarations should be put into a header file (preferably named `new_class.h`) that has to be included in all files that refer to the new class.

It's somewhat bad style (though sometimes necessary) to have a method on Class access variables of SuperClass directly (not by methods). In particular, you should leave the variables of Root (i.e. of struct instance) alone, or you might confuse the Grape kernel.

5.3.4 Adding a Class to the System

Together with the methods described in section 5.4.2 the following methods provide the basic operations for handling classes and methods in an object-oriented environment. All these basic methods belong to class `Root`, therefore they can be invoked on every class via inheritance.

To create a new class call

(CLASS *) GRAPE(superclass, "new-class")(name, size)

> **CLASS *superclass**
> **char *name**
> **int size**

This will create a subclass of `superclass` named `name`. For every new instance of this class `size` bytes of memory will be allocated, therefore `size` should be the size of the struct of the new class (`sizeof(NAME)`). The result of this method is a pointer of type `CLASS *` to the new class. This value should be stored in a class pointer `CLASS *Name`, see section 5.3.2 for naming conventions.

Classes can be deleted by calling

GRAPE(My_Class, "delete")()

> **CLASS *My_Class**

The class `My_Class` will be removed from the class hierarchy only if there are neither subclasses nor instances of `My_Class` left.

All `"delete"` methods should return `NULL`, then the return value of the `"delete"` call can be used to reset the pointer to the instance or class that was deleted:

```
obj = (MY_CLASS *)GRAPE(obj, "delete")();
```

To check if a class with a specific name exists the method

(CLASS *) GRAPE(class, "find-class")(name)

CLASS *class
char *name

can be used. It will search all subclasses of class `class` for a class named `name` and return the pointer to that class or `NULL` if no class with that name exists.

5.3.5 The `Rot2d` Example

Since we want to compute a surface of revolution by rotating a curve around an axis we have to store at least the curve and the resulting triangulation. Storing the axis instead of using an implicit default axis is a good idea, additionally we will need the angle of rotation and the number of segments that should be created.

There are several ways we can implement this class. First of all we could just use `Root` as superclass and add a pointer to a `Triang2d` instance for the surface and two pointers to `Triang1d` instances for curve and axis to it, but we would have to implement many methods to be able to work with this class. The second possibility is to use `Triang1d` as superclass, in this case we would emphasize that we are working with a curve. Since we are ultimately interested in the surface we want to create the best way is probably to implement the new class `Rot2d` as a subclass of `Triang2d`.

We have added curve, axis, rotation angle, discretization and a variable to store the index of the point we are working on (this will be needed later):

```
#define ROT2D_STRUCT                                    \
  TRIANG2D_STRUCT;   /* the superclass is Triang2d */   \
  /* ROT2D */                                           \
  TRIANG1D *curve;   /* the curve that is rotated */    \
  TRIANG1D *axis;    /* axis for the rotation */        \
  double angle;      /* angle of the rotation */        \
  int discr;         /* number of segments to create */ \
  int current        /* current point for editing */


typedef struct {
  ROT2D_STRUCT;
} ROT2D;
```

This definition will be put in the headerfile `rot2d.h`. This headerfile should include declarations like the one for our new class

```
extern CLASS *Rot2d;
```

and things like prototypes for the new methods and functions.

Now our new class has to be added to GRAPE's class hierarchy. First of all we need the class pointer for `Rot2d`, and then we have to call `"new-class"`. We put both into the main program in the file `rotate.c` (the full listing can be found in section 5.5.7):

```
#include <grape.h>
#include "rot2d.h"


CLASS *Rot2d = NULL;


int main()
{
```

```
    Rot2d = (CLASS *)GRAPE(Triang2d, "new-class")("Rot2d", sizeof(ROT2D));
    ASSURE(Rot2d, "main: can't create class Rot2d", return 5);

    ...
  }
```

As you can see the return value of the `GRAPE` call is casted to the type of the result. This cast has to be done explicitly because the `GRAPE` function returns an arbitrary pointer type (this should be `void *`, but in reality it's a `char *` so that you cannot "forget" the cast).

The `ASSURE` macro is used to test if the class could be created, it is provided for error handling and will be described in section 5.4.3.

5.4 Writing Methods

In this section the implementation of new methods is explained in detail. Different types of methods will be presented while the methods needed to make the demo class `Rot2d` work are developed.

5.4.1 Class and Instance Methods

Methods are normal C functions which are registered as methods at runtime. Methods can be send to classes as well as to instances, both types of methods basically have the same format:

```
MY_CLASS *my_class_method_name(parm1, parm2, ...)
{
  MY_CLASS *self;

  self = (MY_CLASS *)START_METHOD(METHOD_TYPE);
  ASSURE(self, "", END_METHOD(NULL));

  ...

  END_METHOD(self);
}
```

The name of the method function is composed of the name of the class the method belongs to followed by a _ and the method name itself.

Class and instance methods are distinguished by the argument of the `START_METHOD` macro. For `METHOD_TYPE` there are three possibilities:

`G_INSTANCE`: The object this method is send to has to be an instance of class `class` or of one of its subclasses. In this case `TYPE` stands for the typedef of the struct defining class `class`.

`G_CLASS`: The method has to be send to class `class` or one of its subclasses. The object returned by `START_METHOD` then has type `CLASS *`. Class methods are for example used to create new instances.

`G_CLASS|G_INSTANCE`: This is the special case of a method that can be send both to a class or an instance. You have to decide yourself if the method was called on a class or an instance by checking the object returned by `START_METHOD` against the class pointer of the object:

```
if(self == (MY_CLASS *)self->class) {
  CLASS *class = (CLASS *)self;
  /* called on class */
  ...
} else {
  /* called on instance */
  ...
```

```
}
```

This type of mixed method should only be used if you really need a method with the same name on both the class and instances of the class.

Writing a method function is like writing a normal function. The only differences are that the object the method is applied to enters the method via `START_METHOD` and is returned by `END_METHOD`. Between these calls the method's object can be manipulated as usual. The parameters of the method call are passed to the method function like to a normal C function, but be aware of the automatic type conversions described at the end of section 5.2.1.

Of course the object of a method may have any valid C name, but calling it `self` makes things easier — you immediatedly know which object you are working on when looking at some part of the method code.

The value returned by `START_METHOD` should be tested, because it could be `NULL` if an instance method was called on a class or vice versa. This can be done with the `ASSURE` macro which is described in section 5.4.3 below. You don't have to print an error message in the case because this will be done by the `START_METHOD` macro, therefore the error string is empty.

A method function may return any pointer with `END_METHOD`, not just the one obtained by `START_METHOD`, an example where these objects differ is the method `"new-instance"`. The return type of a method is defined as the return type of the method function, in case of an error methods always should return `NULL` so that the failure can be detected by the caller.

To avoid trouble with the `GRAPE` function you must start all methods with `START_METHOD`, even if the method's object is not needed for the method. And never use `return` instead of `END_METHOD` to end a method function, this will confuse the internal GRAPE stack.

5.4.2 Adding a Method to the System

Adding and removing methods are basic operations in an object-oriented environment. They are defined on class `Root`, therefore they can be invoked on every class via inheritance.

To add a new method to the system call

(METHOD *) GRAPE(class, "add-method")(name, func)

> **CLASS *class**
> **char *name**
> **void *(*func)()**

This will add a new method to the method list of class `class`. The method is called `name`, `func` is the pointer to the function which implements the new method. The return value of `"add-method"` is the pointer to the structure of the new method, it should only be used to check if the method was successfully added. If the method couldn't be added (maybe there was already a method called `name`), an error message will be printed and `"add-method"` returns `NULL`.

By calling

GRAPE(class, "delete-method")(name)

> **CLASS *class**
> **char *name**

the method `name` can be removed from the method list of class `class`, and with

(METHOD *) GRAPE(class/inst, "find-method")(name)

> **CLASS *class**
> **INSTANCE *inst**
> **char *name**

it can be tested if a method named `name` exists on class `class` resp. on the class of the instance `inst` (of course the superclasses are also searched if necessary). The methods return a `NULL` pointer if the method wasn't found.

5.4.3 Exception Handling

It's important for writing a stable program to do an intensive error checking, especially in an interactive object-oriented environment with features like late binding.

GRAPE provides two general error handling macros

```
ALERT(<expression>, "error message", <statements>)
ASSURE(<expression>, "error message", <statements>)
```

which continue execution only if the <expression> is TRUE. Otherwise ASSURE prints the "error message" string together with the corresponding file name and line number and the user is prompted for a response, ALERT pops up a requester with this information instead if a manager is present. Finally the <statements> are executed.

Usually you should use ALERT because the error message will be shown in the control window. The error message of ASSURE could be hidden if the shell it is printed to is in background, in this case the program would seem to be frozen. But ALERT shouldn't be used in methods on low level classes like devices or interactive classes.

If an error occurs in a method NULL should be returned. This can be done by substituting

```
END_METHOD(NULL)
```

for <statements>. The <statements> are placed in { ;} by the macros. However, if the <statements> contain a comma not surrounded by brackets the C preprocessor thinks to have found too many arguments. It is possible to hide this comma by using another pair of { }.

An example requiring more then one statement: If a method fails after some object obj has been created, then obj can be deleted before NULL is returned:

```
ASSURE(expr,
       "method failed",
       {
         GRAPE(obj, "delete")();
         END_METHOD(NULL);
       });
```

or

```
ASSURE(expr, "method failed",
       GRAPE(obj, "delete")();
       END_METHOD(NULL));
```

Plenty of examples can be found in the methods presented below.

5.4.4 Memory Management

Creating, copying and deleting instances of a class are the basic memory management operations, they should be implemented for every class. Of course they are defined for the base class Root of the GRAPE class hierarchy. First we take a look at allocating and freeing memory, then copying instances will be discussed.

5.4.4.1 "new-instance", "delete" and "free"

The methods "new-instance" and "free" must be written for every new class that adds instance variables to its superclass. If the new class has no additional instance variables the superclass methods can be used via inheritance.

(INSTANCE *) GRAPE(class, "new-instance")(name)

CLASS *class
char *name

first must call "new-instance" on its superclass to allocate memory for a new instance of class class named name. The superclass method will initialize its instance variables and return the pointer to the created instance (again NULL indicates an error). Then the instance variables of the class have to be initialized, finally the pointer to the new instance is returned.

To avoid problems with inheritance the only parameter of a "new-instance" method should be the name of the new instance, only on interactive classes this might be different (see sections 5.5.2 and 9.1). If instances of a class should be archived with the GRAPE archiving methods a different parameter list will get you into trouble — you probably won't be able to read the data back in.

The counterpart of the "new-instance" method is

GRAPE(inst, "delete")()

INSTANCE *inst

This method will delete a reference to an object or — if the object isn't referenced from somewhere else — the object itself, in this case the "free" method is called.

It is not possible to add "delete" methods, this method is only allowed for class Root. If a class needs a special refcount handling a "zero-refs" method has to be implemented, see section 5.4.5.2 below.

(INSTANCE *) GRAPE(inst, "free")()

INSTANCE *inst

This method has to free all memory that was allocated for its instance variables (not only the memory allocated by "new-instance"). By calling "free" on the superclass the superclass' instance variables are freed and finally Root "free" will free the memory for the instance struct.

Of course the superclass "free" method must be called at the end of the method because Root "free" frees the instance itself, afterwards the instance pointer is no longer valid.

"free" must return NULL and it can only be called by "delete", this is done if the instance can be freed since it isn't referenced anymore. Not calling "free" on the superclass, calling it directly or not returning NULL will result in an error.

Now we present the first methods for the demo class Rot2d. The following listing contain lots of detailed comments, it should be no big problem to understand how things are done.

```
ROT2D *rot2d_new_instance(char *name)
{
  CLASS *class;
  ROT2D *self;
  TRIANG1D *curve, *axis;

  /*
   * START_METHOD returns the object the method was called on.
   * "new-instance" is always a class method, therefore G_CLASS
   * is used as parameter.
   */
  class = (CLASS *)START_METHOD(G_CLASS);

  /*
   * The return value is checked because the method could have
   * been called on an instance, then START_METHOD returns NULL.
   * If an error occurs a method always should return NULL, so
   * this is what we do if we get no class.
   */
  ASSURE(class, "", END_METHOD(NULL));
```

```
  /*
   * First we have to allocate memory for our instance, we call
   * "new-instance" on our superclass (using ^). Again we check
   * the return value to see if enough memory was available.
   */
  self = (ROT2D *)GRAPE(class, "^new-instance")(name);
  ASSURE(self, "rot2d_new_instance: can't create instance", END_METHOD(NULL));

  /*
   * We allocate memory for curve and axis by calling "new-instance"
   * on their class Triang1d, again these checks...
   */
  curve = self->curve = (TRIANG1D *)GRAPE(Triang1d, "new-instance")("curve");
  axis = self->axis = (TRIANG1D *)GRAPE(Triang1d, "new-instance")("axis");
  ASSURE(curve && axis,
         "rot2d_new_instance: can't create curve/axis",
         { GRAPE(self, "delete")(); END_METHOD(NULL); });

  /*
   * Only the structure for curve/axis was allocated by "new-instance".
   * We now use "list-alloc" to allocate memory for the coordinate lists,
   * in this case for three/two points. We define our points and set the
   * number of points to the right value.
   */
  GRAPE(curve, "list-alloc")(3);
  curve->x[0] =  0.5; curve->x[1] =  0.5; curve->x[2] =  0.5;
  curve->y[0] = -0.5; curve->y[1] =  0.0; curve->y[2] =  0.5;
  curve->z[0] =  0.0; curve->z[1] =  0.0; curve->z[2] =  0.0;
  curve->number_of_points = 3;

  GRAPE(axis, "list-alloc")(2);
  axis->x[0] =  0.0; axis->x[1] =  0.0;
  axis->y[0] = -1.0; axis->y[1] =  1.0;
  axis->z[0] =  0.0; axis->z[1] =  0.0;
  axis->number_of_points = 2;

  /*
   * The other instance variables get default values, too. The
   * default point for editing is the middle point of the curve.
   */
  self->angle = 360.0;
  self->discr = 24;

  self->current = 1;

  /*
   * We return the instance we have created.
   */
  END_METHOD(self);
}


ROT2D *rot2d_free()
{
  ROT2D *self;

  /*
   * This is an instance method, therefore G_INSTANCE is used.
```

```
   * Again the return value is checked, somebody might have called
   * "free" on the class Rot2d (or a subclass).
   */
  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * First we have to delete curve and axis...
   */
  if(self->curve)
    GRAPE(self->curve, "delete")();
  if(self->axis)
    GRAPE(self->axis, "delete")();

  /*
   * Now calling "free" on the superclass will free any memory
   * allocated for the surface of revolution (this is done by
   * Triang2d "free") and the memory for the instance struct
   * (Triang2d "free" calls "free" on its superclass, too).
   */
  GRAPE(self, "^free")();

  /*
   * Free methods should always return NULL...
   */
  END_METHOD(NULL);
}
```

New methods have to be added to their class by `"add-method"`. Therefore we extend the main program, this has to be done for all following methods (the full listing can be found in section 5.5.7):

```
#include <grape.h>
#include "rot2d.h"


CLASS *Rot2d = NULL;


int main()
{
  Rot2d = (CLASS *)GRAPE(Triang2d, "new-class")("Rot2d", sizeof(ROT2D));
  ASSURE(Rot2d, "main: can't create class Rot2d", return 5);

  GRAPE(Rot2d, "add-method")("new-instance", rot2d_new_instance);
  GRAPE(Rot2d, "add-method")("free", rot2d_free);

  ...
}
```

5.4.4.2 `"list-alloc"` and `"list-free"`

You might have noticed that the `"new-instance"` method of `Rot2d` used a method called `"list-alloc"` to allocate memory for the coordinate lists of the curve and axis triangulations.

Most geometry classes have got the methods `"list-alloc"` and `"list-free"` for allocating and freeing memory for their pointer type instance variables, for example coordinates lists, connectivity array or even instances.

GRAPE(inst, "list-alloc")(...)

INSTANCE *inst

allocates memory for the pointer variables of the instance `inst`. The parameters of this method depend upon the class of the instance, you should be aware of the problems that are created by these different parameter lists. If the method is called on an instance of a subclass, there might be parameters missing or parameter types may be wrong. Sometimes a `G_CAST_CLASS` to cast the object to the right class might help to ensure that the call is done correctly.

For all geometry classes the functionality of `"list-alloc"` is now identical, the sizes of the lists will be changed to exactly the given new sizes. You have to make sure that the currently used space (for triang class the `number_of_points`/`number_of_elements`) doesn't exceed the new maximum size. These `"list-alloc"` methods use the smart realloc functions described in section 10.2.1, therefore existing data is preserved if possible.

GRAPE(inst, "list-free")()

INSTANCE *inst

frees all memory that was allocated by `"list-alloc"` for the pointer type instance variables. This method needs no parameters, those classes providing `"list-free"` usually call it in their `"free"` method.

For geometry classes calling `"list-free"` is identical to setting the size of the used space to zero and then calling `"list-alloc"` with zero sizes, e.g. for `TRIANG2D *tr2d`:

```
tr2d->number_of_points = tr2d->number_of_elements = 0;
GRAPE(tr2d, "list-alloc")(0, 0);
```

This is not recommended because it may cause trouble for subclasses, you should always use `"list-free"` if it is available.

We could have written these methods for our class `Rot2d` though we don't have any memory lists. In this case `"list-alloc"` would have been used to create default curve and axis, `"list-free"` to free them again. Then `"delete"` would have called `"list-free"` to delete curve and axis instead of deleting them itself.

5.4.5 Copying Instances

There are two fundamentally different ways an instance can be copied: the instance itself can be copied to create a completely new instance within its own memory, or the pointer to the instance can be copied to create a reference. In the main manager menu buttons for both copy types are supplied (see section 4.3.5.3).

5.4.5.1 Copy Methods

First we discuss copying the instance itself. There are two copy methods which are implemented for most geometry classes.

GRAPE(inst, "hardcopy")()

INSTANCE *obj

creates an identical copy of `inst`. All variables, lists and instance entries of `inst` are copied to a new instance which is returned. This method is also available in the interactive environment (see 4.3.5.3).

GRAPE(inst, "softcopy")(copy)

INSTANCE *inst, *copy

creates a copy of **inst** in an existing instance **copy**, if **copy** is **NULL** a new instance is allocated. Differences to **"hardcopy"** are that storage available in **copy** is used, new storage is only allocated if necessary, and in case new storage has to be allocated only the exact amount of storage needed is allocated. In general both objects are therefore not identical, they are only identical from a geometric point of view.

We could use the superclass method to copy our **Rot2d** instances, this would copy the surface but not axis and curve. Therefore we have implemented these two method, too.

```
ROT2D *rot2d_hardcopy()
{
  ROT2D *self, *copy;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * First we call "hardcopy" on the superclass Triang2d to copy
   * the coordinate and connectivity lists.
   */
  ASSURE(copy = (ROT2D *)GRAPE(self, "^hardcopy")(),
         "rot2d_hardcopy: can't copy superclass", END_METHOD(NULL));

  /*
   * The superclass method has called "new-instance", therefore we
   * have these default curves, but we don't need them. We delete
   * them and reset the pointer to be sure it is valid.
   */
  GRAPE(copy->curve, "delete")();
  copy->curve = NULL;
  GRAPE(copy->axis, "delete")();
  copy->axis = NULL;

  /*
   * Now if there is a curve it is copied by calling "hardcopy"
   * and assigning the result to the curve of the copy. The
   * result is checked to see if we really got a copy.
   */
  if(self->curve)
    ASSURE(copy->curve = (TRIANG1D *)GRAPE(self->curve, "hardcopy")(),
           "rot2d_hardcopy: can't copy curve",
           { GRAPE(copy, "delete")(); END_METHOD(NULL); });

  /*
   * Same for the axis.
   */
  if(self->axis)
    ASSURE(copy->axis = (TRIANG1D *)GRAPE(self->axis, "hardcopy")(),
           "rot2d_hardcopy: can't copy axis",
           { GRAPE(copy, "delete")(); END_METHOD(NULL); });

  /*
   * The other values only have to be copied.
   */
  copy->angle = self->angle;
  copy->discr = self->discr;

  copy->current = self->current;
```

```
  END_METHOD(copy);
}


ROT2D *rot2d_softcopy(ROT2D *copy)
{
  ROT2D *self;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * First call "softcopy" on the superclass Triang2d. This will create
   * a new instance if necessary and copy the lists.
   */
  ASSURE(copy = (ROT2D *)GRAPE(self, "^softcopy")(copy),
         "rot2d_softcopy: can't copy superclass", END_METHOD(NULL));

  /*
   * Now we copy the curve. Since we might use the default curves memory
   * of the copy we delete it only if the original instance has no curve.
   */
  if(self->curve) {
    copy->curve = (TRIANG1D *)GRAPE(self->curve, "softcopy")(copy->curve);
    ASSURE(copy->curve, "rot2d_softcopy: can't copy curve",
           { GRAPE(copy, "delete")(); END_METHOD(NULL); });
  } else {
    GRAPE(copy->curve, "delete")();
    copy->curve = NULL;
  }

  /*
   * Same for the axis.
   */
  if(self->axis) {
    copy->axis = (TRIANG1D *)GRAPE(self->axis, "softcopy")(copy->axis);
    ASSURE(copy->axis, "rot2d_softcopy: can't copy axis",
           { GRAPE(copy, "delete")(); END_METHOD(NULL); });
  } else {
    GRAPE(copy->axis, "delete")();
    copy->axis = NULL;
  }

  /*
   * The other values only have to be copied.
   */
  copy->angle = self->angle;
  copy->discr = self->discr;

  copy->current = self->current;

  END_METHOD(copy);
}
```

5.4.5.2 Refcount Handling

The information about references to an instance is stored in the instance itself — this is what the instance variable `refcount` of the base class `Root` is used for. If no reference to an instance exists (that is the instance is only used once), the `refcount` is zero, for each reference created it is incremented and for every reference deleted it is decremented. As explained above an instance can only be deleted if there are no references left. For checking if there are references `"delete"` uses the method

GRAPE(inst, "zero-refs")()

INSTANCE *inst

This method must return `NULL` if there are still references left and a different value (usually `inst`) if the instance `inst` can be deleted. `Root "delete"` calls `"zero-refs"`, if it returns `NULL` only the `refcount` is decremented otherwise `"free"` is called.

For most classes you don't have to write a `"zero-refs"` method, because `Root "zero-refs"` (which just checks the `refcount` and returns the appropriate value) can be used, only for classes with cyclic references like `TimeStep` or `Geom2d` a special `"zero-refs"` method is needed.

Deleting references must be handled by the `"delete"` method, to simplify creating references the macro

```
ASSIGN(<ref>, <inst>)
```

is provided. Here both `<ref>` and `<inst>` are pointers to an instance struct, the macro will copy the pointer `<inst>` to `<ref>` and increment the `refcount` of the instance.

Never try to copy an instance by copying the C structure (of course this can be used to copy datatypes which are structs like `COMPLEX`). This will get you into big trouble, because it destroys the instance list of the class the instance belongs to. To copy instances the copy methods described in section 5.4.5.1 can be used.

5.4.6 Display Methods

Another important category of methods are display methods. You should be familiar with the general interaction strategy of the GRAPE manager (see section 4.3.2): there is always a "current display method" which is used to update the graphics window. This method can be changed interactively, the default display method is `"display"`. Therefore for a new geometry class you should at least implement this default display method, of course the `"display"` method of the superclass can be used via inheritance instead.

Display methods (beside `"display"`) should always have the suffix `"-disp"`, they must be parameterless. It is possible to use interactive elements like rulers to provide parameters for a display method as it is done in the example below. This will be explained in section 5.5.5.

As mentioned we could use the superclass method to display the `Rot2d` instances, but we would only see the surface of revolution and not the curve we are working on or the axis. Therefore we implement a `"display"` method for our new class, it draws curve, axis and surface and the point we are currently working on (the index of this point is stored in the instance variable `current`). For drawing the point an icosahedron is used, its "radius" can be controled by a ruler.

```
ROT2D *rot2d_display()
{
  ROT2D *self;
  TRIANG1D *curve;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));
```

```
/*
 * Before calling "display" on the curve to show it, we have
 * to check if we really have a curve with points. A curve with
 * no points would be no problem, but then the "display" is not
 * necessary. No curve is a problem, we would get an error
 * message "NULL object for method 'display'".
 */
if((curve = self->curve) && curve->number_of_points) {
  GRAPE(curve, "display")();

  /*
   * We want to see the point we edit, therefore we have to display
   * it in some way. Drawing a single point doesn't help much, there-
   * fore we draw a small icosahedron around the point. But first
   * lets check if there is a point in the curve with index current.
   */
  if(self->current >= 0 && self->current < curve->number_of_points) {
    /*
     * The ruler is defined static and initialized with NULL so that
     * we can detect if it was already created. The variable iparm
     * defines the "midpoint" and the "radius" for the icosahedron.
     */
    static double iparm[4] = { 0.0, 0.0, 0.0, 0.05 };
    static RULER *radius = NULL;
    MANAGER *mgr;

    /*
     * We have to create the ruler the first time this method is
     * called. It get the name and label "point radius" and the
     * variable it controls is the radius in iparm. The other
     * values define the layout of the ruler.
     */
    if(!radius)
      radius = (RULER *)new_item(Ruler, I_Name, "point radius",
                            I_Var, &(iparm[3]), dfDouble,
                            I_MinMax, 1.e-5, 0.25, I_Scale, 0.25,
                            I_Offset, 0.0, I_StepSize, 0.025, I_End);

    /*
     * The standard manager is the object that controls the two
     * windows you get when grape is started.
     */
    mgr = (MANAGER *)GRAPE(Manager, "get-stdmgr")();

    /*
     * We have to tell the manager that we want our ruler displayed
     * when this method is active. We used "new-handle" to tell the
     * manager that the object put into the manager by "add-inter"
     * that follows the "new-handle" belongs to rot2d_display. The
     * '1' indicates that we include one object into the manager.
     */
    if(radius && mgr && GRAPE(mgr, "new-handle")(rot2d_display, 1))
      GRAPE(mgr, "add-inter")(radius);

    /*
     * Now copy the coordinates of the current curve point to the
     * array that is used to get an icosahedron at the position
```

```
         * and with the radius defined by iparm.
         */
        iparm[0] = curve->x[self->current];
        iparm[1] = curve->y[self->current];
        iparm[2] = curve->z[self->current];

        /*
         * get_icosaeder returns a Triang2d instance which we can
         * display with the standard "display" method.
         */
        GRAPE(get_icosaeder(iparm), "display")();
      }
    }

    /*
     * As for the curve we only call "display" on the axis if we
     * really have one.
     */
    if(self->axis && self->axis->number_of_points)
      GRAPE(self->axis, "display")();

    /*
     * Upto now we have only drawn the axis and the curve. To draw
     * the surface itself we just call "display" on the superclass
     * Triang2d.
     */
    GRAPE(self, "^display")();

    /*
     * Everything was ok, so we return the instance.
     */
    END_METHOD(self);
  }
```

In this example we were able to use already existing display methods (`"display"` on `Triang1d` and `Triang2d`) to draw the objects. If you want to draw lines and/or (shaded) patches yourself the standard graphicdevice (the graphic window is an instance of class `GraphicDevice`) can be used, see section 9.2.2.

5.4.7 Methods for Debugging

There are several kinds of debug methods providing information about classes and instances. Some of them can be used interactively, for example to pick points or objects in the graphic window to get information about them.

5.4.7.1 Debugging Class and Instance Lists

These following methods print some information about a class to the standard output. They can't be used interactively (unless a method is supplied which calls them), because they have to be send to a class.

GRAPE(class, "print-classes-send")()

CLASS *class

prints the names of all subclasses of class `class`,

GRAPE(class, "print-instances-send")()

CLASS *class

prints the names of all instance of class `class` and

GRAPE(class, "print-methods-send")()

CLASS *class

prints the names of all methods of class `class`.

5.4.7.2 Debugging Instances

For some projects the following methods are pre-installed on buttons.

GRAPE(inst, "show-send")()

INSTANCE *inst

exists on many classes, it prints nearly all data of the instance `inst` like coordinate and connectivity vectors to the standard output. This allows debugging the data.

GRAPE(surf, "debug")()

SURFACE *surf

currently exists only on class `Surface` (and its subclasses). The user may pick for example parts of the instance or a single point with the mouse and obtain a lot of information.

5.4.7.3 The Methods "get-info" and "follow"

If you have worked with the GRAPE manager before or read section 4.3.5.3 about the hierarchy management in GRAPE you may have asked yourself, how the information list you get when you click on the current instance is created, and how the instances around the current instances are determined. This is done by the following two methods.

GRAPE(inst, "get-info")(list)

INSTANCE *inst
XLIST *list

The method can add as many strings containing information about the instance as wanted to the given list by calling the method `"add-to-list"` on `list` with a string as parameter, but first it should call `"get-info"` on the superclass to collect the information about it in the list.

For our `Rot2d` example we only add the rotation angle and the discretization to the list, all information about curve and axis can be obtain by changing the current instance of the manager (see below).

```
ROT2D *rot2d_get_info(XLIST *list)
{
  ROT2D *self;
  char str[MAX_STRLEN];

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * Most of the work is done by the superclass method. It puts
   * information about the name, class, refcount, number of points
   * and so on into the list.
   */
  GRAPE(self, "^get-info")(list);

  /*
```

```
   * Just add the angle and the discretization to the list. We
   * also could display information about axis and curve but
   * the user can get this information by changing the object
   * to axis or curve and then clicking on it.
   */
  sprintf(str, "angle: %f", self->angle);
  GRAPE(list, "add-to-list")(str);
  sprintf(str, "discr: %d", self->discr);
  GRAPE(list, "add-to-list")(str);

  END_METHOD(self);
}
```

The environment around the current instance of the manager is determined by the method

GRAPE(inst, "follow")(obj, pop_push)

INSTANCE *inst
INSTANCE *obj[3][3]
char pop_push[3][3]

Each entry in the three by three arrays represents one of the three by three objects in the manager. First `"follow"` must be called on the superclass to initialize the arrays, then instance variables of `inst` can be put in the `obj` array. For each instance that is put into the array a character has to be put into the `pop_push` array, available are:

- `' '` No instance, this is the default set by `Root "follow"`.
- `'d'` Go down: the current instance has to move left.
- `'n'` Like `next_scene`: the current instance has to move up.
- `'p'` Instance is pushed, pop it to go there.
- `'c'` Next element in a doubly linked list like timesteps. Since this instance will show the current instance in its `"follow"`, don't push the current instance.

Don't worry if you don't understand this, it's really a bit complicated. Just look at the example for class `Rot2d`, this might help to understand how the method works.

```
ROT2D *rot2d_follow(INSTANCE *inst[3][3], char pop_push[3][3])
{
  ROT2D *self;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * The inst and pop_push arrays define which objects will be
   * shown in the manager if a Rot2d instance is the current
   * object (there are three by three instances possible).
   * In this case the superclass method just initializes the
   * array with zeros, it does not provide its own environment.
   */
  GRAPE(self, "^follow")(inst, pop_push);

  /*
   * If we have axis and/or curve we want is to be shown top right
   * and right of the Rot2d instance. The `d' stands for `down',
   * this means that the Rot2d instance is moved to the left if
   * we click on the axis or curve.
```

```
   */
  if(self->axis) {
    inst[0][2] = (INSTANCE *)self->axis;
    pop_push[0][2] = 'd';
  }
  if(self->curve) {
    inst[1][2] = (INSTANCE *)self->curve;
    pop_push[1][2] = 'd';
  }

  END_METHOD(self);
}
```

5.4.8 The "universal" Method

As described in section 5.2.1 `GRAPE` searches for a method `"universal"` if the method that is send to an instance is neither found in the instance's class nor in its superclasses. If the method

GRAPE(inst, "universal")(method)

> **INSTANCE *inst**
> **char *method**

is found, it is send to the instance with the method `method` that was originally called as parameter. If you supply an `"universal"` method for a class the method `method` can for example be passed on to one of the instance variables of the instance `inst`.

Be careful: only parameterless methods can be called via universal, any argument passed to the original method is lost. Therefore if the method `method` needs parameters on the class it is send to this won't work. Because the `GRAPE` function doesn't know if a method needs parameters it will always try to use `"universal"` if the original method wasn't found.

The behaviour of the `GRAPE` function can be influenced by calling one of the methods

GRAPE(obj, "unset-universal-send")()
GRAPE(obj, "set-universal-send")()

> **INSTANCE/CLASS *obj**

`"unset-universal-send"` switches the universal mode off, `"set-universal-send"` switches it on again (by default it is on). Be careful when switching universal mode off, many classes like `Scene`, `TimeScene` and `Time_Object` rely on this mechanism.

5.4.9 More Methods for Rot2d

Now we have written the memory management, copy and display methods for our class `Rot2d` and set up its interactive environment. Now we will write the main method which computes the surface of revolution from the curve, rotation angle and discretization. Watch how existing methods and utility functions are used to simplify the method.

```
ROT2D *rot2d_rotate_send()
{
  ROT2D *self;
  TRIANG1D *curve, *axis;
  MATRIX44 rotseg, rot, prod;
  VEC3 paxis[2], prot;
  int number_of_points;
  int number_of_elements;
  int ulines, vlines;
  int i, j, ind;
```

```
double angle;

self = (ROT2D *)START_METHOD(G_INSTANCE);
ASSURE(self, "", END_METHOD(NULL));

/*
 * Before we start computing the surface some check have to be done.
 * Curve and axis must exist and have enough resp. the right number
 * of points. Creating a negative or zero number of segments doesn't
 * make much sense and we need a non-zero angle.
 */
ASSURE((curve = self->curve) && curve->number_of_points > 1,
       "rot2d_rotate_send: no curve with enough points",
       END_METHOD(NULL));
ASSURE((axis = self->axis) && axis->number_of_points == 2,
       "rot2d_rotate_send: no correct axis found",
       END_METHOD(NULL));
ASSURE(self->discr > 0,
       "rot2d_rotate_send: need discr > 0",
       END_METHOD(NULL));
ASSURE(self->angle != 0.0,
       "rot2d_rotate_send: need angle != 0",
       END_METHOD(NULL));

/*
 * First the number of points and elements is computed. For discr
 * segments we have discr + 1 lines, the number of lines in the
 * other direction is given by the curve.
 */
ulines = self->discr + 1;
vlines = curve->number_of_points;

/*
 * A grid with each quadrilateral divided into two triangles
 * is used, so we need these numbers of points an elements:
 */
number_of_points = ulines * vlines;
number_of_elements = 2 * (ulines - 1) * (vlines - 1);

/*
 * Now check if this Rot2d instance has enough memory to store
 * the new triangulation. We compare both the computed number
 * of points and triangles to the maximum number available.
 */
if(self->max_number_of_points < number_of_points ||
   self->max_number_of_elements < number_of_elements)
  /*
   * If there isn't enough memory we just extend the old memory
   * lists by calling "list-alloc". We will overwrite the old data
   * which is preserved by "list-alloc" later.
   */
  GRAPE(self, "list-alloc")(number_of_points, number_of_elements);

/*
 * The information about how much of the memory of the instance
 * is used has to be stored in the instance, "list-alloc" only
 * sets the maximum number of points and elements to the array
 * sizes.
```

```
 */
self->number_of_points = number_of_points;
self->number_of_elements = number_of_elements;

/*
 * Each segment of the surface is angle / discr wide. g_line_rotate
 * computes the rotation matrix for a rotation around and arbitrary
 * axis. Unfortunately we have to copy the end points of the axis
 * because this function needs an array of two VEC3. The other para-
 * meters are the angle and the matrix that has to be computed.
 * The rotseg matrix describes the rotation for one segment, the
 * rotation matrix rot is set to the identity matrix because we
 * start with no rotation.
 */
angle = (2.0 * M_PI / 360.0) * self->angle / (double)self->discr;
paxis[0][0] = axis->x[0]; paxis[0][1] = axis->y[0]; paxis[0][2] = axis->z[0];
paxis[1][0] = axis->x[1]; paxis[1][1] = axis->y[1]; paxis[1][2] = axis->z[1];
g_line_rotate(paxis, angle, rotseg);
g_matrix44_set_identity(rot);

/*
 * Now we compute the points. For each segment was have to transform
 * the points of the curve with the current rotation matrix. Again
 * we have to copy the points from and to the triangulation to use
 * the matrix multiplication function (alternatively we could have
 * done the multiplication by hand). After each segment the rotation
 * matrix is multiplied with the segment rotation matrix to get the
 * matrix needed for the next segment.
 */
for(ind = 0, i = 0; i < ulines; i++) {
  for(j = 0; j < vlines; j++, ind++) {
    prot[0] = curve->x[j]; prot[1] = curve->y[j]; prot[2] = curve->z[j];
    g_matrix44_mult_vec3(rot, prot);
    self->x[ind] = prot[0]; self->y[ind] = prot[1]; self->z[ind] = prot[2];
  }
  g_matrix44_mult(prod, rotseg, rot);
  g_matrix44_assign(prod, rot);
}

/*
 * We also have to define the connectivity of the triangulation
 * (which points belong to which triangle). Again we run through
 * the segments storing two triangles for each quadrilateral.
 * It's really quite simple in this case...
 */
for(i = 0; i < ulines - 1; i++)
  for(j = 0; j < vlines - 1; j++) {
    ind = (i * (vlines - 1) + j) * 2;
    self->vertex[ind][0] = i * vlines + j;
    self->vertex[ind][1] = (i + 1) * vlines + j;
    self->vertex[ind][2] = i * vlines + j + 1;
    ind++;
    self->vertex[ind][0] = i * vlines + j + 1;
    self->vertex[ind][1] = (i + 1) * vlines + j;
    self->vertex[ind][2] = (i + 1) * vlines + j + 1;
  }

/*
```

```
   * Some methods also need information about the neighbours of the
   * triangles. Computing this directly is no problem, but since
   * we have a method for this we just call it.
   */
  GRAPE(self, "make-neighbour-send")();

  END_METHOD(self);
}
```

Creating the surface is one part of the job, but first we have to create a curve we can rotate (the default curve is not very interesting...). The interactive editing is described in the next section, here we present the methods for inserting points in and deleting them from a **Triang1d** instance. The methods could be made more efficient by using the system function **memmove**, but this way they are easier to understand (they won't be called too often with too many points anyway).

```
TRIANG1D *triang1d_insert_point(int number)
{
  TRIANG1D *self;
  int i;

  self = (TRIANG1D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * First some checks. Since we insert the new point at position
   * "number", this index can not be the first point, it has to be
   * in [1, number of points - 1].
   */
  if(number <= 0 || number > self->number_of_points - 1) {
    fprintf(stderr,
            "triang1d_insert_point: can't insert point at position %d\n",
            number);
    END_METHOD(NULL);
  }

  /*
   * If we have already used all points in this triang1d instance we
   * need new memory for the new point. "list-alloc" will do the job
   * because for Triang1d it will copy the coordinates to the new
   * arrays, therefore we lose no information and get more memory!
   * Be careful: there are still some "list-alloc" methods (e.g. on
   * List_Of_Inst) which will not preserve the old data.
   */
  if(self->number_of_points == self->max_number_of_points)
    GRAPE(self, "list-alloc")(self->max_number_of_points + 1);

  /*
   * We have to shift the points from index number on to get a free place
   * for storing the new point (memmove would probably be faster)
   */
  for(i = self->number_of_points - 1; i >= number; i--) {
    self->x[i + 1] = self->x[i];
    self->y[i + 1] = self->y[i];
    self->z[i + 1] = self->z[i];
  }

  /*
   * The new point is interpolated from the two points next to it,
   * its just the midpoint of the line between these points
```

```
   */
  self->x[number] = (self->x[number - 1] + self->x[number + 1]) / 2.0;
  self->y[number] = (self->y[number - 1] + self->y[number + 1]) / 2.0;
  self->z[number] = (self->z[number - 1] + self->z[number + 1]) / 2.0;

  /*
   * Now we have an additional point...
   */
  self->number_of_points++;

  END_METHOD(self);
}


TRIANG1D *triang1d_delete_point(int number)
{
  TRIANG1D *self;
  int i;

  self = (TRIANG1D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * Again first the checks. We don't want curves with no points or just
   * one point, and we have to make sure that the index number is valid.
   */
  if(self->number_of_points <= 2) {
    fprintf(stderr, "triang1d_delete_point: less than three points left\n");
    END_METHOD(NULL);
  }
  if(number < 0 || number > self->number_of_points - 1) {
    fprintf(stderr, "triang1d_delete_point: "
                    "can't delete point at position %d\n", number);
    END_METHOD(NULL);
  }

  /*
   * Like in "insert-point" we shift the points from index number
   * on, but now in the other directions. And we have one points less.
   */
  for(i = number; i < self->number_of_points - 1; i++) {
    self->x[i] = self->x[i + 1];
    self->y[i] = self->y[i + 1];
    self->z[i] = self->z[i + 1];
  }
  self->number_of_points--;

  END_METHOD(self);
}
```

The next two methods are the class `Rot2d` counterpart to the `"insert-point"` and `"delete--point"` methods on class `Triang1d`. They can be called interactively (we will put them on a button) because they have no parameters, this is not possible for the `Triang1d` methods. For the moment you should ignore the calls to `"update-point"`, this method will be introduced in the next section.

```
ROT2D *rot2d_insert_point()
{
  ROT2D *self;
```

```
  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * We definitely need a curve to insert a point.
   */
  if(!self->curve || !self->curve->number_of_points)
    END_METHOD(NULL);

  /*
   * We just call "insert-point" on the curve with the current
   * point as parameter to insert the point. Now the coordinate
   * rulers should show the coordinates of the new point,
   * therefore we update them.
   */
  if(GRAPE(self->curve, "insert-point")(self->current))
    GRAPE(self, "update-point")();

  END_METHOD(self);
}


ROT2D *rot2d_delete_point()
{
  ROT2D *self;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * We need a curve for delete, too.
   */
  if(!self->curve || !self->curve->number_of_points)
    END_METHOD(NULL);

  /*
   * Just like insert, again we have to update the rulers,
   * because we have changed the current point.
   */
  if(GRAPE(self->curve, "delete-point")(self->current))
    GRAPE(self, "update-point")();

  END_METHOD(self);
}
```

Now we have all we need to create and edit `Rot2d` instances, but this can up to now only be done by programming. To be able to work interactively we have to add an interface to the methods and variables to the GRAPE manager.

5.5 Interactive Elements

Before we explain how interactive elements like buttons and rulers can be created and added to the GRAPE manager permanently or depending on a display method, we will give a short overview of what objects exist and what they are used for.

5.5.1 What is Available

The base class for all elements in the GRAPE control window is the abstract class `Item`, a subclass of `Root`. The static elements like texts and color rectangles are subclasses of `Item`, the variable elements like rulers are subclasses of the abstract class `Interactive`, a subclass of `Item`.

There exists many different interactive types in GRAPE which can be used to control variables or to send methods to different objects (the number given with each element refers to its documentation in chapter 9.1):

- `Buttons` (9.1.3.2) are used to send a method to an instance or class. The instance can be set explicitly, if no special instance was given the method of the button will be send to the manager's current instance.
- `Checkbox`es (9.1.3.4) are a special kind (a subclass) of `Button`, they can be toggled between two states and are used to switch flags. If a whole set of flags has to be handled a `CheckField` (9.1.3.3) — a collection of `Checkbox`es — can be used.
- `Input`s (9.1.3.13) are the simplest elements for controlling an integer, float, double or string variable, the variable can be changed by typing in the new value.
- `Ruler`s (9.1.3.22) are a more comfortable way to control a variable. Additional to an input field they contain elements for changing the variable with the mouse.
- `Function_Ruler`s (9.1.3.10) are time-depended rulers. Their variable is controlled by a spline which can be edited with the `Spline_Editor` (9.1.3.26) that is associated to the function-ruler.
- `FileLists` (9.1.3.8) are used to select a file from the list of files in the current directory matching a given wildcard.

As you might have noticed while working with GRAPE there are much more interactive elements, some of them subclasses of the ones mentioned above, like `Slider` (9.1.3.24) which is a subclass of `Ruler` with a different look but basically the same functionality. Other interactive elements are composed of several (simple) elements, they can be very complex but nevertheless they are easy to use — like `Color_Selector`s (9.1.3.5), which allow editing a color given by RGB values in RGB or HSV mode, or `Matrix_Editor`s (9.1.3.18) which control a transformation matrix. The basic classes for these composite elements are `Group` (9.1.3.12) and `Layer` (9.1.3.16).

Most of these interactives were already described in the introduction in section 4.1, the full documentation of all interactive classes and their methods can be found in chapter 9.1.

5.5.2 Creating Items

Items are created like all other instances by sending `"new-instance"` to their class. For items this method needs different parameters depending on the class, we will use a button which sends the method `"show-send"` to the current instance and a ruler which controls a double variable as examples:

```
BUTTON *button;
RULER *ruler;
static double *var = 0.5;

button = (BUTTON *)GRAPE(Button, "new-instance")("show-send", NULL, "show");
ruler = (RULER *)GRAPE(Ruler, "new-instance")(&var, "var", dfDouble);
```

The button is called "show", the `NULL` indicates that it has no special instance to send its method `"show-send"` to, therefore it will send it to the current instance.

The ruler is named "var" like the variable it controls, it has type `dfDouble`. The variable was declared `static` and it was initialized, you should be very careful when you are using local or uninitialized variables for rulers (see section 5.5.8 below).

Items have many different attributes that can be changed: name, label, size, position, color, method, instance, action, minimum, maximum, scale, fillmode etc. — what is possible depends on the class. To set for example the width of the button to 6.0 and its height to 1.0 you have to send `"set-pref-size"` to the button:

```
GRAPE(button, "set-pref-size")(6.0, 1.0);
```

Sending the methods

```
GRAPE(ruler, "set-min-max")(0.0, 1.0);
GRAPE(ruler, "set-scale")(1.0);
GRAPE(ruler, "set-offset")(0.0);
```

restricts the ruler value to the interval [0.0, 1.0] and changes its layout so that the ruler value is 0.0 in the left and 1.0 in the right position. As mentioned before the documentation of all these methods can be found in chapter 9.1.

5.5.3 The `new_item` Function

Creating items and setting their attributes can get quite complex, especially when groups or layers are involved. Of course you have to remember the order of parameters for the `"new-instance"` methods and which methods have to be called first — for example you have to call `"set-offset"` after setting minimum, maximum and scale, otherwise the result is undefined. To simplify creating complex items we have created an interface which takes care of the mentioned problems and does an extended error checking. It consists of a single function with a variable parameter list:

```
ITEM *new_item(CLASS *, ...);
```

The first parameter is the class of the item which should be created, then the item description follows. If the item can be created it is returned. With this function the sequence of GRAPE calls above for creating the ruler can be replaced by the single call

```
ruler = (RULER *)new_item(Ruler, I_Name, "var", I_Var, &var, dfDouble,
                          I_MinMax, 0.0, 1.0, I_Scale, 1.0, I_Offset, 0.0, I_End);
```

The item description consists of so called "tags" (all starting with `I_`) and their parameters. It always has to be terminated by `I_End`, otherwise you will get errors and the item will not be created.

The `new_item` function is currently under developement therefore some classes like checkfields and function-rulers are still missing. An example on how this function can be used is given in section 5.6.6.1.

5.5.3.1 The `new_item` Tags

The following list describes all tags with their parameters, for which classes they are allowed or required and which methods the tags are related to. The tags are sorted by class, starting with the basic class `Item` followed by its subclasses. Of course the tags are always valid for all subclasses of the listed classes. Again for a detailed explanation of the class hierarchy (and the methods the tags are related to) you should consider the handle documentation in chapter 9.1.

You might just scan through this rather long section to see what's available or skip reading it since it serves as a reference and it not necessary for understanding the rest of the chapter.

`I_Name, char *name (Item "set-name")`: Define a name for the item. If no name is given then for interactives the label, method or action and for textmessages the text is used instead. If none of these is given the item will be named according to its class and you will get a warning (because it doesn't make much sense to create for example a button which is labeled "button" and sends the method `"button"` when it is pressed).

I_Size, double sizex, double sizey (Item "set-pref-size"): Set the size of the item. If you only want to set the width or height of the item you can use `I_SizeX` or `I_SizeY` instead. If there is no place for the item in the group it is added to it will not be visible until you make the group big enough. For groups with follow up groups (like the option menus) the behaviour is different: all groups will be checked until the item fits into one of the groups. If `MENU_MAX_SIZE` is specified, the item will be as wide/high as the group it is added to (only if there is enough free place), if the group size is changed the item is scaled accordingly.

I_SizeX, double sizex (Item "set-pref-size"): Only set the width of the item. The item gets the default height 1 which should be ok for buttons, rulers, sliders, inputs etc. For some items like groups you should also set the height using `I_Size` instead.

I_SizeY, double sizey (Item "set-pref-size"): Only set the height of the item. For the width of the item the group size is used therefore it will change its size if the group is resized. If the item should have a fixed width you should you `I_Size` instead.

I_RSize, double fracx, double fracy (Item "set-rel-size-mode"/"set-pref-size"): Define the size of the item relative to the group it is in. `fracx` and `fracy` define the fractions of the group width and height that will be used for the item. If the item is placed in a layer then resizing the layer will also scale the item. To fill a group with items which have relative sizes, the `fracx` and `fracy` values should add up to 1. `I_RSize` is very useful for items in groups with border and/or title, because the border and title size don't have to be taken into account when selecting item sizes.

I_RSizeX, double fracx (Item "set-rel-size-mode"/"set-pref-size"): Only set the relative width of the item (see `I_RSize` about relative sizes). The default height is 1 as for `I_SizeX`.

I_RSizeY, double fracx (Item "set-rel-size-mode"/"set-pref-size"): Only set the relative height of the item (see `I_RSize` about relative sizes). The default width is the size of the group the item is in, see `I_SizeY`.

I_Pos, double posx, double posy (Item "set-pref-pos"): Set the position of the item. Positions are calculated from the upper left corner of the group the item belongs to, they are given in "normalized" coordinates (the default width of the control device window is 12). Be careful: (`posx`, `posy`) is the position of the *lower* left corner of the item, therefore if the item has height 2 and should have a distance of 3 from the top of the group then you have to use 5 for `posy`. You should avoid defining many items with fixed positions but use groups instead.

I_PosX, double posx (Item "set-pref-pos"): Only set the horizontal position of the item. The vertical position will be calculated automatically. See `I_Pos` on how positions are calculated.

I_PosY, double posy (Item "set-pref-pos"): Only set the vertical position of the item. The horizontal position will be calculated automatically. See `I_Pos` on how positions are calculated.

I_Color, VEC3 rgb (Item "set-color"): Set the color of the item. You should avoid using direct rgb colors because the number of colors for the control window is limited. If you really need colored items you should use `I_ColorIndex` instead, then the already predefined colors are used.

I_ColorRGB, double r, double g, double b (Item "set-color"): Same as `I_Color` with the same restrictions, but the rgb values can be given separately.

I_ColorIndex, int index (Item "set-color-index"): Choose a color for the item from the predefined control window colors. These colors are

color name	rgb values
BLACK_COLOR	(0.0, 0.0, 0.0)
RED_COLOR	(1.0, 0.0, 0.0)
GREEN_COLOR	(0.0, 1.0, 0.0)
YELLOW_COLOR	(1.0, 1.0, 0.0)
BLUE_COLOR	(0.0, 0.0, 1.0)
MAGENTA_COLOR	(1.0, 0.0, 1.0)
CYAN_COLOR	(0.0, 1.0, 1.0)
WHITE_COLOR	(1.0, 1.0, 1.0)

The colors `FOREGROUND_COLOR`, `BACKGROUND_COLOR`, `PRESSED_LOWER_COLOR`, `PRESSED_UPPER-_COLOR`, `TEXT_COLOR`, `LINE_COLOR`, `LIGHT_ON_COLOR` are used for drawing the items in the control window, their values depend on the system.

`I_FillMode, int fmode (Item "set-fill-mode")`: Define how the item should be placed within its group if no position was set with `I_Pos(X|Y)`. `MENU_FILL_TOP` will place the item as far top and left and `MENU_FILL_BOTTOM` as far bottom and right as possible. Layers can be placed relative to the mouse position with `MENU_MOUSE_RELATIVE`: in this case the position set with `I_Pos(X|Y)` is used to determine the offset of the lower left corner of the layer relative to the mouse. Default fillmode if `MENU_FILL_BOTTOM` for textmessages and rulers and `MENU_FILL_TOP` for all other items.

`I_AddAction, char *add, char *remove (Item "set-add-action")`: The add-action `add` resp. remove-action `remove` is called on the item every time it becomes visible resp. invisible, for example when the menu is changed. A visible item may be hidden (temporarily) by open layers, this will not result in a call to the add- or remove-action. If a layer has been bound to the item with `"layer-list-add"` and `"add-layers"` is used for `add` and `"remove-layers"` for `remove`, then the layer is only visible in the item's menu.

`I_HelpText, char *text (Item "set-help-text")`: Set the help text for an item that is shown when the middle mouse button is pressed while holding the control key. This text is intended to serve as a short online-documentation of what happens when the item is used.

`I_State, int state (Button "set-state")`: This will set the state of a button to `PRESSED` or `UNPRESSED`. Since the default state of a button is `UNPRESSED` only setting it to `PRESSED` makes sense.

`I_DrawMethod, char *draw (Button "set-draw-method")`: Set the method that will be used to draw the button. For example the method `"draw-arrow"` is used to draw ruler and scrollbar arrows. If you want to write your own draw method you should consider the handle documentation.

`I_Align, ALIGN_FLAGS align (Textmessage "set-align")`: Define how the text of the textmessage should be aligned. `afTop`, `afCenter` and `afBottom` can be used for vertical, `afLeft`, `afCenter` and `afRight` for horizontal alignment. The default value is `afLeft|afCenter`.

`I_Text, char *text (Textmessage "set-text")`: Set the text of the Textmessage. The text is copied to an internal buffer therefore you have to use this method everytime you want to change the message. A copy of the text can be obtain with the `"get-text"` method. The message can be aligned using `I_Align`, default is left aligned and vertically centered. The text may consist of several lines separated by `\n` (like in printf).

`I_Action, char *action (Interactive "set-action")`: The action is send to the interactive when it gets some input, for example when a pressed button is released or when the return key is pressed while an input is active. For buttons the default action sends the method set by `I_Method` to the instance set by `I_Instance` or the current instance of the manager if none was set, for all other interactives the method is send to the interactive itself. Before writing you own action method you should consider the handle documentation on what an action has to do for a specific class.

I_Instance, INSTANCE *inst (Interactive "set-instance"): Define to which instance the method of the interactive (set by `I_Method`) should be send. By default the method is send to the manager's current instance.

I_Self, (Interactive "set-instance"): This is a special case which can't be handled by `I_Instance`: the interactive is set as its own instance. This if often used for buttons if their method can't be send to the current object because it provides some special functionality not depending on a specific class. Most buttons in the manager menu use `I_Self`, the mark button for example sends `"mark"` to itself to store the current instances as marked instance in a local variable.

I_Method, char *method (Interactive "set-method"): Set the method that is send by the interactive's action (see `I_Action` to the interactive itself or its instance set by `I_Instance` resp. the manager's current instance if none was set.

I_Label, char *label (Interactive "set-label"): Define a label for the interactive (the text that is printed onto the object). If no name was given then the label is used as name, too. For inputs the label can be controlled by setting a print-format with `I_PrintFormat`.

I_Redraw, REDRAW_FLAG redraw (Interactive "set-redraw-flag"): If the redraw flag is set to `rfOn` (default value), then every time an action is caused by the interactive the graphics window will be refreshed by redrawing the current object. If this redraw is not necessary (as for example for the dismiss buttons in some layers) the redraw flag should be set to `rfOff` to avoid this refresh.

I_Inactive, (Interactive "set" INACTIVE): Usually an item is active when it is created. If you need an item that is only active under certain conditions, e.g. if the current object of the manager has a special class, this tag can be used to inactivate the item.

I_SideEffect, INTERACTIVE *effect (Interactive "add-side-effect"): Side effects define how interactives depend on each other. For example the input field and the ruler in a bar1d (which show the value of the same variable) define each other as side effects. When the variable is changed by one of these items then `"mark-side-effect"` is called which sends `"update"` to all side effects, this will update the value that is shown by the other one.

I_BarColor, VEC3 rgb (Bar1d "set-bar-color"): Set the bar color of the item. You should avoid using direct rgb colors because the number of colors for the control window is limited. Use `I_BarColorIndex` instead if possible.

I_BarColorRGB, double r, double g, double b (Bar1d "set-bar-color"): Same as `I_BarColor` with the same restrictions, but the rgb values are given separately.

I_BarColorIndex, int index (Bar1d "set-bar-color-index"): Choose a color for the bar of the bar1d from the predefined control window colors. See `I_ColorIndex` for a list of these colors.

I_EvalFn, double (*eval)(double), double (*ieval)(double) (Bar1d "set-eval-fn"): The eval functions define how the variable the bar1d controls depends on the value that is shown on the bar1d. By default `g_eval_ident_fn` is used, therefore the bar1d shows exactly the variable value. By specifying `g_eval_exp_fn` and `g_eval_log_fn` you may create an exponentially or logarithmically scaled bar1ds. Another predefined function is `g_eval_angle_fn`, which is for example used by the rotation rulers in the matrix editor to limit the angle value to $[-180, 180]$. Of course you may define your own evaluation functions.

I_MinMax, double min, double max (Bar1d "set-min-max"): Limit the value for the bar1d to [min, max]. If min > max then the limits are ignored. Don't try to use both I_Min and I_Max instead of I_MinMax, this will not work because I_Min and I_Max also set a maximum resp. minimum value. By pressing the right mouse button while the mouse is over the bar of the bar1d you get a configuration layer which allows to change these values and offset, scale and stepsize interactively (see section 4.1.2.5).

I_Min, double min (Bar1d "set-min-max"): Define the minimum value for the bar1d. The upper limit will be set to the maximum default value 10^{37}. See I_MinMax for more information.

I_Max, double max (Bar1d "set-min-max"): Define the maximum value for the bar1d. The lower limit will be set to the minimum default value -10^{37}. See I_MinMax for more information.

I_FixMinMax, (Bar1d "fix-min-max" (ON, ON)): By default the minimun and maximum values of a bar1d can be changed using the bar1d's config layer, forbidding to change these values is useful since it ensure that the value stays within a predefined range. If you want to ensure for example that a bar1d value is always non-negative you can use I_Min, 0.0 and I_FixMinMax.

I_Offset, double offset (Bar1d "set-offset"): The offset and the scale (set by I_Scale) of the bar1d define the layout of its ruler. The scale is the difference between the ruler values at the left (bottom) and right (top) position and offset defines a "starting value": the values of the ruler in its end positions will be offset + n scale.

I_Scale, double scale (Bar1d "set-scale"): Set the scale for the bar1d. See I_Offset on how scale and offset determine the layout of the bar1d.

I_StepSize, double stepsize (Bar1d "set-step-size"): The variable of the bar1d will be decremented resp. incremented by stepsize when the arrows next to the ruler or slider are pressed. Pressing these arrows will always (apart from limits) result in a value that is a multiple of stepsize, therefore if the current value is 0.5 and stepsize is 1.0, after the first click on the incrementing arrow the value is 1.0, a second click will give 2.0 and so on. By the way, these arrows have an auto-repeat feature — just keep them pressed.

I_ValueChanged, char *vchanged (Bar1d "set-value-changed"): Define a method that is set to the bar1d every time its value is changed. This method is used to distinguish between interactive changes of the ruler (clicking into the ruler or on an arrow or entering a value in the input field) and non-interactive changes (changing the variable of the ruler by a function/method or as a side-effect). For interactive changes this method is called by the bar1d's action (see I_Action).

I_PrintFormat, char *pformat (Bar1d/Input "set-print-format"): Set a format string which is used to print the value onto the input field if it is not active. This pformat is used in an sprintf with the value of the input as first and the label as second argument. Additional to the normal printf formatting options a number n followed by $ is allowed as modifier, this will select the nth argument for this format. On SGI machines this is done by the standard printf function, on other systems this is emulated.

I_Var, void *var, DECLARATION_FORMAT df[, int maxsize] (Bar1d/Input "set-var-and-type"): This tag is required — bar1ds and inputs don't make much sense without a variable. The first argument is the pointer to the variable, its type is defined by the second argument. For bar1ds the types dfInt, dfFloat and dfDouble are allowed, for inputs also dfString is possible. For inputs you have to specify the maximum length of the input field as third argument, if dfString is used this length should match the length of the var string.

`I_Var, int *var (CheckField/Checkbox "set-variable"):` The `I_Var` tag is also required for CheckFields and Checkboxes. These classes always use an integer variable therefore only the one `var` argument is needed.

`I_TabAction, char *taction (Input "set-tab-action"):` The method `taction` will be send to the input when the tab key is pressed. At the moment this is only used for the input field in lists to allow the completion of the input (like the command completion in the tcsh).

`I_Eval, char *eval, char *inv_eval (Input "set-eval"):` The evaluation method is used to convert the input string into a variable value, the inverse method is used for the conversion in the other direction. Don't use `I_Eval` unless you know exactly what you are doing, because these method have to handle print formats and if the input is active or not.

`I_MaskMode, int mode (CheckField "set-mask-mode"):` Set the mask mode of a checkfield. Values allowed for `mode` are `G_CHECKFIELD_RADIO` (only one checkbox in the checkfield can be selected at a time), `G_CHECKFIELD_CBOXES` (checkboxes are independed, but switch different bits in the checkfield's variable) or `G_CHECKFIELD_CUSTOM` for custom defined checkfields.

`I_NewBox, CHECKFIELD_VAR id, CHECKBOX *box (CheckField "insert-checkbox):` Add a new checkbox to a checkfield. `id` is used to determinate which bits of the checkfield's variable should be switched. The `id`'s of different checkboxes of the checkfield should access different bits of the checkfield variable. Of course you can use `new_item(Checkbox, ...)` to create the `box`.

`I_Type, int type (Line "set-type"):` For lines the types `LINE_FLAT` and `LINE_ELEVATED` are allowed. You should avoid using lines to separate items and use groups instead if possible.

`I_Item, ITEM *item (Group "add-inter"):` Add an item to the group. Of course this item can be a group itself, it also be created by `new_item` (the function may be called recursively).

`I_Border, BORDER_FLAGS bflag (Group "change-border"):` Select if the border (`bfBorder`) or title (`bfTitle`) or both (`bfBorder|bfTitle`) of the group will be drawn, default is `bfNoBorder` for groups and `bfBorder|bfTitle` for layers. Be careful: switching on the border or title will reduce the interior size of the group, the border takes up 0.5 in each direction, the title 0.75 in vertical direction.

`I_FixSize, int fixx, int fixy (Layer "fix-size"):` With `ON` for `fixx/fixy` the layer size can be fixed horizontally/vertically to its predefined size. By default the size is not fixed (`OFF` for both `fixx` and `fixy`), the layer then can be sized with the mouse (see section 4.1.2.1).

`I_ButtonList, BUTTON_DESCR *bdescr (Group ):` The tags `I_ButtonList`, `I_RulerList` and `I_SliderList` should only be used for converting old projects (they correspond to `P_ButtonList`, `P_RulerList` and `P_SliderList`, they makes it possible to group the buttons, rulers or sliders together in one group. Be careful with `I_ButtonList`: for `P_ButtonList` the button instances may be uninitialized (a pointer to a class defined in the same project can be used), for `I_ButtonList` this is not allowed.

`I_RulerList, RULER_DESCR *rdescr (Group ):` See `I_ButtonList`.

`I_SliderList, SLIDER_DESCR *sdescr (Group ):` See `I_ButtonList`.

5.5.4 Adding Items to the Manager

An item has to be added to the control window of the manager before it can be used, therefore you first have to get the pointer to the GRAPE manager by sending the message `"get-stdmgr"` to the class `Manager`:

```
MANAGER *mgr = (MANAGER *)GRAPE(Manager, "get-stdmgr")();
```

This method will create the manager if it does not exist, otherwise the pointer to the existing manager is returned. This method can fail if the windows cannot be opened therefore it is wise to check the return value at least when the method is called for the first time (for the `Rot2d` example this is done in the `"get-editor"` method).

User defined items are always placed in one of the option menus. With

```
GRAPE(mgr, "add-inter")(inter);
```

it is placed in the first option menu, with

```
GRAPE(mgr, "add-inter-for-opt")(inter, opt_number);
```

it will be added to the option menu with number `opt_number` (`"add-inter"` is just a shortcut for `"add-inter-for-opt"` with zero `opt_number` as parameter).

Where the item is placed depends on its attributes like for example the position (default or set with `"set-pref-pos"`/`I_Pos`) and/or fill-mode (default or set with `"set-fill-mode"`/`I_Fill-Mode`) and the other items that are already present in the option menus. First the manager tries to place it in the option menu and at the position the user wanted. If this fails another position in the same menu will be tried, if this is also impossible the next option menu is used instead. If the item cannot be added to one of the option menus the methods return `NULL`.

Items can be added to a group or layer instead of the manager itself, in this case the `"add-inter"` method has to be send to the group:

```
GRAPE(group, "add-inter")(item);
```

Of course now the `group` has to be added to the manager.

We won't go into details about the concepts of the GRAPE manager now, again we refer to chapter 9.1.

5.5.5 Display Methods Using Items

Parameters for a display method can be provided with interactives, in most cases by variables that are controlled by rulers. An example for this was already given in section 5.4.6: the display method of class `Rot2d` draws an icosahedron around the point that is currently edited, to control the radius of the icosahedron a double variable controlled by a ruler is used.

Of course items in display methods also have to be added to the manager with `"add-inter"` or `"add-inter-for-opt"`, but for these methods this only has to be done when the method is active, and the items have to be removed when the method is no longer used. To control adding and removing these display method dependent items the method

```
GRAPE(mgr, "new-handle")(display_method_function, number_of_items)
```

is used. `display_method_function` is the pointer to the function that implements the display method and `number_of_items` is the number of items that are added to the manager by this method. The method returns `TRUE` (a pointer that is not `NULL`) if the items have to be added and `FALSE` (a `NULL` pointer) if they were already added.

In the `Rot2d` example the lines to insert the ruler look like this:

```
if(GRAPE(mgr, "new-handle")(rot2d_display, 1))
  GRAPE(mgr, "add-inter")(radius);
```

When the display method gets active the `"new-handle"` method returns `TRUE` and the ruler is added. Now while it is active `"new-handle"` always returns `NULL` and nothing happens. When the method is no longer used the manager automatically removes all items that belonged to that method, therefore they can be added again the next time the method is activated.

Of course all kinds of items can be used with `"new-handle"`, even groups and layers. Items that belong together should be put into a group, then they always appear in the same option menu. Layers added by display methods are not restricted to one of the option menus, they are visible in all manager menus.

5.5.6 A User-Interface for Rot2d

What do we need for working with `Rot2d` instances? First of all buttons for sending the methods `"rotate"`, `"insert-point"` and `"delete-point"` are needed, additionally we create a `"reset"` button which resets the surface to its initial state. The rulers for `angle`, `discr`etization and `current` point are no problem, but editing the point is more difficult: the rulers will be bound to the current point coordinate variables, but when the current point is changed these rulers have to be bound to new variables. Therefore we first provide a method for changing the coordinate ruler variables:

```
/*
 * The coordinate rulers are the most difficult part of the user interface,
 * because the point that is edited can be changed (by a ruler). We need
 * them in the method "update-point", so we make them global.
 */
static RULER *xpos, *ypos, *zpos;


ROT2D *rot2d_update_point()
{
  ROT2D *self;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * Again: check if there is a curve. In this case it is very
   * important to test if current is a valid index, because
   * we used it to access the coordinate array.
   */
  if(!self->curve || self->current < 0 ||
     self->current > self->curve->number_of_points - 1)
    END_METHOD(NULL);

  /*
   * Now bind the coordinate ruler to the coordinates of the
   * (new) current point. They will be updated automatically.
   */
  GRAPE(xpos, "set-variable")(&self->curve->x[self->current]);
  GRAPE(ypos, "set-variable")(&self->curve->y[self->current]);
  GRAPE(zpos, "set-variable")(&self->curve->z[self->current]);

  END_METHOD(self);
}
```

The method `"set-variable"` is sent to the rulers to update their variables, of course we first have to check if the current point is valid.

We use a method to create the buttons and rulers, otherwise the main program would get rather long. The rulers are bound to the `Rot2d` instance the method is send to:

```
ROT2D *rot2d_get_editor()
{
  ROT2D *self;
  MANAGER *mgr;
  RULER *current, *angle, *discr;
  BUTTON *rotate, *reset, *insert, *delete;
```

```
self = (ROT2D *)START_METHOD(G_INSTANCE);
ASSURE(self, "", END_METHOD(NULL));

/*
 * Again these checks... But try to remove them and set current
 * to something like -10000, this will surely crash the program.
 * We need a curve, points and a valid current value.
 */
if(!self->curve || !self->curve->number_of_points ||
   self->current < 0 || self->current > self->curve->number_of_points - 1)
  END_METHOD(NULL);

/*
 * Like in Rot2d "display" we need the manager for adding
 * our rulers and buttons.
 */
ASSURE(mgr = (MANAGER *)GRAPE(Manager, "get-stdmgr")(),
       "rot2d_get_editor: can't get mgr", END_METHOD(NULL));

/*
 * First we create rulers for the current point, the coordinates
 * of the current point, the angle and the discretization. The
 * coordinate rulers have to be updated when the current point
 * is changed, see below.
 */
current = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->current, "point", dfInt);
xpos = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->curve->x[self->current], "x pos", dfDouble);
ypos = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->curve->y[self->current], "y pos", dfDouble);
zpos = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->curve->z[self->current], "z pos", dfDouble);
angle = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->angle, "angle", dfDouble);
discr = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->discr, "discr", dfInt);

/*
 * Now the buttons, the don't get any special object, so they
 * will send their method to the current object. This might be
 * dangerous, because someone can change to the curve or axis
 * and send "insert-point" or "delete-point" to it by pressing
 * the button. What about the parameter in this case?
 */
rotate = (BUTTON *)GRAPE(Button, "new-instance")
  ("rotate-send", NULL, "rotate");
reset = (BUTTON *)GRAPE(Button, "new-instance")
  ("reset", NULL, "reset");
insert = (BUTTON *)GRAPE(Button, "new-instance")
  ("insert-point", NULL, "insert-point");
delete = (BUTTON *)GRAPE(Button, "new-instance")
  ("delete-point", NULL, "delete-point");

/*
 * Creating the rulers and buttons has never
 * failed before, but who nows...
 */
```

```
    ASSURE(current && xpos && ypos && zpos && angle && discr &&
           rotate && reset && insert && delete,
           "rot2d_get_editor: can't create ruler/button", END_METHOD(NULL));

    /*
     * This is the most interesting part. We tell the current ruler
     * to call "update-point" on the Rot2d instance when it is changed
     * by the user. This method will bind the coordinate rulers to
     * the new current point, see below.
     */
    GRAPE(current, "set-instance")(self);
    GRAPE(current, "set-method")("update-point");

    /*
     * We define a suitable size for the buttons - half the width
     * of the standard control window. The rulers default size is ok.
     */
    GRAPE(rotate, "set-pref-size")(6.0, 1.0);
    GRAPE(reset, "set-pref-size")(6.0, 1.0);
    GRAPE(insert, "set-pref-size")(6.0, 1.0);
    GRAPE(delete, "set-pref-size")(6.0, 1.0);

    /*
     * Finally add all items to the option menu of the manager.
     */
    GRAPE(mgr, "add-inter")(current);
    GRAPE(mgr, "add-inter")(xpos);
    GRAPE(mgr, "add-inter")(ypos);
    GRAPE(mgr, "add-inter")(zpos);
    GRAPE(mgr, "add-inter")(angle);
    GRAPE(mgr, "add-inter")(discr);
    GRAPE(mgr, "add-inter")(rotate);
    GRAPE(mgr, "add-inter")(reset);
    GRAPE(mgr, "add-inter")(insert);
    GRAPE(mgr, "add-inter")(delete);

    END_METHOD(self);
}
```

The most interesting part are the lines where the `current` ruler is configured to send the method `"update-point"` to the `Rot2d` instance whenever it it changed (with the mouse or by typing in a new value):

```
GRAPE(current, "set-instance")(self);
GRAPE(current, "set-method")("update-point");
```

This ensures that the coordinate rulers always point to the right variables.

5.5.7 Putting It All Together

Now we have all methods we need to work with `Rot2d` instances. Our main program has to add the new class and its methods, to create a `Rot2d` instance to work on and the editor for this instance. Finally the manager is started by sending the `"handle"` method and we can work interactively with the instance we have created.

```
#include <grape.h>
#include "rot2d.h"
```

```
/*
 * Define the new class pointer, it is initialized below
 */
CLASS *Rot2d = NULL;


int main()
{
  MANAGER *mgr;
  ROT2D *rot2d;

  /*
   * First we call "new-class" to create our class Rot2d. The method
   * is send to the superclass, its parameters are the name and the
   * size of the structure, this size will be used by "new-instance"
   * to allocate memory for the instance.
   */
  Rot2d = (CLASS *)GRAPE(Triang2d, "new-class")("Rot2d", sizeof(ROT2D));
  ASSURE(Rot2d, "main: can't create class Rot2d", return 5);

  /*
   * Now add the methods for the new class and the two methods
   * we have written for the existing class Triang1d. "add-method"
   * is called on the class the method is defined on, parameters
   * are the method name and the pointer to the function. This
   * pointer is used when GRAPE()() is called to call the function
   * corresponding to the method name.
   */
  GRAPE(Rot2d, "add-method")("new-instance", rot2d_new_instance);
  GRAPE(Rot2d, "add-method")("free", rot2d_free);
  GRAPE(Rot2d, "add-method")("hardcopy", rot2d_hardcopy);
  GRAPE(Rot2d, "add-method")("softcopy", rot2d_softcopy);
  GRAPE(Rot2d, "add-method")("rotate-send", rot2d_rotate_send);
  GRAPE(Rot2d, "add-method")("display", rot2d_display);
  GRAPE(Rot2d, "add-method")("get-info", rot2d_get_info);
  GRAPE(Rot2d, "add-method")("follow", rot2d_follow);
  GRAPE(Rot2d, "add-method")("get-editor", rot2d_get_editor);
  GRAPE(Rot2d, "add-method")("update-point", rot2d_update_point);
  GRAPE(Triang1d, "add-method")("insert-point", triang1d_insert_point);
  GRAPE(Triang1d, "add-method")("delete-point", triang1d_delete_point);
  GRAPE(Rot2d, "add-method")("insert-point", rot2d_insert_point);
  GRAPE(Rot2d, "add-method")("delete-point", rot2d_delete_point);

  /*
   * Now we can create our first Rot2d instance, it is called "rot".
   * This could fail so we test the return value.
   */
  rot2d = (ROT2D *)GRAPE(Rot2d, "new-instance")("rot");
  ASSURE(rot2d, "main: can't create object", return 5);

  /*
   * Now we bind the rot2d variables to the rulers and
   * create the buttons that will send our methods.
   */
  ASSURE(GRAPE(rot2d, "get-editor")(), "main: can't create editor", return 5);

  /*
   * Since we want to work with our Rot2d instance interactively
```

```
     * we have to start the manager. This is done by calling "handle"
     * with the instance as parameter. The control will be returned
     * to this function when the exit button is pressed.
     */
    mgr = (MANAGER *)GRAPE(Manager, "get-stdmgr")();
    GRAPE(mgr, "handle")(rot2d);

    return 0;
}
```

The method for archiving `Rot2d` instances is not explained here but in the section about the GRAPE XDR archiving system, see 5.7.6.

All files for the demo program can be found in the directory `demo/rot2d/basic` of the GRAPE distribution, the main file is called `rotate.c`. If you don't know where the GRAPE files are located ask the system operator or the person who installed GRAPE on your computer.

When the program is started the `rot2d` instance passed to the manager in the `"handle"` method is the current instance. Top right and right of it you will see axis and curve, this is what the method `"follow"` was written for. The buttons and rulers were installed in the first option menu.

5.5.8 Problems of the Interface

When working with the `Rot2d` program you will notice that the buttons were added from the top and the rulers from the bottom of the menu (this is the default), therefore the rulers are "upside down". It would be better to group all buttons and rulers together and to rearrange them, this would make life easier.

The values of the `current` ruler are not limited, therefore we can change it to an invalid value. Without the error check in the methods `"update-point"`, `"insert-point"` and `"delete-point"` the program would not be very stable.

We forgot to add the suffix `"-send"` to the `Rot2d` methods for inserting and deleting points. This would be no problem if we hadn't put the method on buttons, now we can call the equally named `Triang1d` methods by just changing the current object to axis or curve and pressing one of the buttons. But these method need a parameter therefore the result is undefined.

The biggest problem is created by the way the rulers are bound to the `Rot2d` instance. It is possible to delete the instance within the manager (just press the expand button in the main menu, then click on the `Rot2d` instance and delete it with one of the delete buttons), then the pointer for the ruler variables and instance will point to nowhere. Just try it and look at the ruler values, if you are lucky the program will not crash if you edit them. A solution for this problem would be to put the rulers into the `Rot2d` structure, then they could be deleted with the instance.

Only one `Rot2d` instance can be edited because the rulers are explicitly bound to it. Of course we can call `"get-editor"` on a second instance, but what would happen? We would get a second set of rulers, but we cannot decide which rulers belong to which instance. We would get a second set of buttons even though we don't need them. Even worse, the pointers to the coordinate rulers would be overwritten, calling `"update-point"` would always update the rulers for the second instance (again this problem could be solved by putting the rulers in the `Rot2d` structure).

Most of the problems can be solved easily, this will be done when we create a project out of this demo program in the next section.

5.6 Projects

What projects are and how they are used interactively was already explained in section 4.5. Now we will describe how to create new projects, as an example we make a project out of the demo program presented in the last section.

5.6.1 How Projects Work

First we give a short overview on how projects work internally, this will be discussed in detail in the following sections.

The basic functions the class `Project` provides for managing projects are `"setup"`, `"use"`, `"unuse"`, `"add"` and `"remove"`. The user must provide a project specific setup-method, she/he may provide init-, exit- and add-methods for the project which are called by the corresponding method on class `Project`.

The basic method for adding the classes, methods and items of a project is `"use"`. When called for the first time `"use"` calls the user defined setup-method of the project. This setup-method declares which classes, methods and init-, exit- and add-method belong to the project and which projects the project depends upon, and it creates the items that should be managed by the project. `"use"` first calls `"use"` on all projects the project depends upon, then — if the project wasn't already in use — classes, methods and item are added and the init-method is called.

The method `"add"` is used when a project is added by selecting it from the manager's project list (the one you get by pressing the "Project: add" button). `"add"` first calls `"use"`, then the add-method of the project. The add-method can define an object which will be put in the data hierarchy, finally information about the project and its author can be displayed.

The counterpart to `"add"` is `"remove"`. The only thing `"remove"` does is to call `"unuse"` which, if the project isn't needed anymore, calls the exit-method and removes everything that was added by `"use"` in reverse order (items, methods, and other projects, classes are not removed for several reasons).

5.6.2 Creating Projects

Projects are created like all instances by sending

GRAPE(Project, "new-instance")(name, char *setup, hidden)

> **char *name**
> **char *setup**
> **int hidden**

to the class. `name` is the name of the project to be created, it has to be unique therefore `"new-instance"` will fail if a project with the same `name` already exists.

`setup` is the name of the project's setup method, it determines which classes, methods, etc. will be managed by the project. Writing a project setup method will be explained in the next section.

If the `hidden` flag is `TRUE` the project will not be included in the manager's project list. This way projects can be hidden from the users view, nevertheless hidden projects can be added by typing in their name. There are methods `"hide"` and `"show"` which allow to hide projects or to make them visible again after they have been created, but these methods should only be used internally.

A more elegant way to declare new projects is provided by the function

> **void g_project_include(PRTYPE *);**

which takes a `NULL` terminated list of projects

```
typedef struct {
  char *name;      /* name of the project */
  char *setup;     /* setup function name */
  void *(*func)(); /* setup function pointer */
  int hidden;      /* hidden flag */
} PRTYPE;
```

as parameter. The setup methods will be added automatically by the `g_project_include` function, if `"new-instance"` is used the setup methods have to be added with `"add-method"`. In section 5.6.7 we will present a "standard" GRAPE main program which uses this function to add the user project list.

Project can't be deleted, therefore `Project`'s `"free"` method will display an error message when you accidentally try to delete a project.

5.6.3 The "setup" Method

A setup-method has to be provided for every project (everything else is optional), it should be called `"pname-setup"` where `pname` is the name of the project. When a project is used for the first time this setup-method is called to initialize it. For declaring which other projects, classes, methods, items, etc. belong to the project the method

GRAPE(project, "setup")(...)

PROJECT *project

has to be sent to it within the setup-method. `"setup"` has a variable parameter list (like the `new_item` function), each parameter starts with a tag (with prefix `P_`) followed by its arguments. The list has to be terminated with `P_End` otherwise the project initialization will probably fail.

The order of the tags is only significant if the same tag is used several times: if for example a class and a subclass of this class should be created first the class and then its subclass have to be given, because they are created in this order. The `"use"` method will always add methods after the classes have been added, therefore methods can be given before the class they belong to is declared.

The following list describes all possible tags and their parameters. Some of the tags will be used in our `Rot2d` example in section 5.6.6.2.

`P_Project, char *name:` The name of a project this project depends upon. The `"(un)use"` method will send `"(un)use"` to project `name` to add (remove) it before this project is added (removed). Be careful: currently loops are not allowed, therefore none of the `"use"`d projects can `"use"` this one again.

`P_ProjectList, PROJECT_DESCR *plist:` Same as `P_Project`, but now the whole `NULL` terminated list of list of project is added (removed). Again, loops are not allowed. `PROJECT_DESCR` is defined as

```
typedef struct {
  char *name;
} PROJECT_DESCR;
```

`P_Class, CLASS **super, CLASS **class, char *name, int size:` Create a new class `name` and store the class pointer in `class` (therefore the pointer pointer). `size` has the same meaning as in the `"new-class"` method (section 5.3.4), it should be the size of the structure defining the instances of the class. The new class will be a subclass of `super`. Here a pointer to the superclass-pointer has to be provided because the superclass might be created within the project, in this case the superclass-pointer is not sufficient (it would be `NULL`).

`P_ClassList, CLASS_DESCR *clist:` Same as `P_Class` for a list of classes. Each entry in `clist` has the same structure as the parameters to `P_Class`:

```
typedef struct {
  CLASS **superclass;
  CLASS **class;
  char *name;
  int size;
} CLASS_DESCR;
```

This list has to be `NULL` terminated, too.

`P_ClassCN, char *classname, CLASS **class, char *name, int size:` This tag is used to create a new class even if the class pointer of the superclass is unknown. `classname` is the name of the superclass, the other parameters are the same as for `P_Class`. This tag makes only sense for dynamic shared object and shouldn't be used yet.

`P_Method, CLASS **class, char *name, void *(*func)():` Add the method `name` implemented by the method function `func` to class `class`. Again a class pointer pointer is needed, because the class might be created by the project.

`P_MethodList, METHOD_DESCR *mlist:` Same as `P_Method` for a list of method. Each entry in `mlist` has the same structure as the parameters to `P_Method`:

```
typedef struct {
  CLASS **class;
  char *name;
  void *(*func)();
} METHOD_DESCR;
```

Like all project lists this one has to be `NULL` terminated. This is the list that is most often used (for example in our `Rot2d` example in section 5.6.6.2). Since every method has its own class (pointer) pointer methods from different classes can be handled with one list.

`P_MethodCN, char *classname, char *name, void *(*func)():` As `P_ClassCN` this tag only makes sense for dynamic shared objects and shouldn't be used yet. It is used to add a method without knowing the class pointer, the name of the class `classname` is used instead. The other parameters are identical to those of `P_Method`.

`P_Item, ITEM *item:` The `item` will be handled by the project. The items of a project are only visible if the project is in use, they will be added to and removed from the manager automatically with the project. If the items are visible depends on the batch flag that is given as parameter to `"use"`. When the project is added from the manager's project list the batch flag is always `FALSE` and the items are added. The `item` is added with `"add-inter"`, therefore it is placed in the first option menu.

`P_ItemOpt, ITEM *item, int menu_nr:` Same as `P_Item`, but the item will be placed in option menu `menu_nr`. There is no tag "P_ItemList" because items can be and should be grouped.

`P_ButtonList, BUTTON_DESCR *blist:` This and the following tags are provided for compatibility with the old project management where it was only possible to handle lists of buttons, rulers and sliders. Each entry in `blist` has the format

```
typedef struct {
  char *methname;     /* method to be send by the button */
  INSTANCE **obj;     /* instance to send it to, NULL if current instance */
  char *txt;          /* label of the button */
  double l, h, x, y;  /* size and position */
  int optmenu;        /* option menu the button should be added to */
} BUTTON_DESCR;
```

Again a pointer pointer to the instance of the ruler has to be provided, because the instance of the ruler could be a class that is created by the project. The rulers will be created by the `"setup"` method, nevertheless the instances are initialized later when the items are added. The buttons of the list can be placed in a group by using the `new_item` function's tag `I_ButtonList` (see section 5.5.3), but in this case the button's instances must be initialized.

`P_RulerList, RULER_DESCR *rlist:` Same as `P_ButtonList` for rulers, see there. `RULER_DESCR` has the format

```
typedef struct {
  void *var;          /* variable modified by the ruler */
  char *txt;          /* label of the ruler */
  double x, y;        /* position of the ruler */
  char *action;       /* method called when the ruler is modified */
  int type;           /* ruler's type (dfInt, dfFloat, dfDouble) */
  int optmenu;        /* option menu the ruler should be added to */
} RULER_DESCR;
```

P_SliderList, SLIDER_DESCR *slist: Same as P_ButtonList for sliders, see there. SLIDER_-DESCR is defined as

```
typedef struct {
  void *var;          /* variable modified by the slider */
  char *txt;          /* label of the slider */
  double x, y;        /* position of the slider */
  char *action;       /* method called when the slider is modified */
  int type;           /* slider's type (dfInt, dfFloat, dfDouble) */
  int optmenu;        /* option menu the slider should be added to */
} SLIDER_DESCR;
```

Because of the fixed structure of the lists the possibilities when creating buttons, rulers or sliders with P_ButtonList, P_RulerList or SliderList are very limited. These tags should really only be used if an old project is converted.

P_InitMethod, char *init: Set the init-method of the project. This method has to be added by the project with P_Method or within a P_MethodList.

P_ExitMethod, char *exit: Set the exit-method of the project. This method has to be added by the project, too.

P_AddMethod, char *add: Set the add-method of the project. Has to be added, too.

P_Version, int version, int revision: With P_Version, P_Date and P_Author a standard message can be set for the project. The first line contains the method name, the second the version and date and the third one the author information. The message will be displayed every time the project is "add"ed.

P_Date, char *date: Used together with P_Version and P_Author, see P_Version.

P_Author, char *author: Used together with P_Version and P_Date, see P_Version.

5.6.4 Using or Adding a Project

All classes, methods and items of a projects are added by the method

GRAPE(project, "use")(batch)

PROJECT *project
int batch

This is of course only done if the project isn't already in use. The batch flag determines if the items should be used, too. If the items should be supressed because the program is in global batch mode and the manger isn't available the value of g_batch_mode should be used for batch, for a description of the global batch mode and the g_batch_mode variable see 9.1.4. All projects the project depends upon are "use"d first, then the project's objects are added and finally a user defined init-method, which should be called "pname-init" for project pname, is called when the project goes into use.

How does a project know that it is in use? Everytime the method "use" is called the project's use count is increased. If the use count is zero the project isn't in use, then the classes, methods and items have to be added and the init-method is called. When "unuse" (see below) is called the use count is decreased and if it is zero all the objects added by "use" are removed after the exit-method has been called. The projects the project depends upon are "(un)use"d every time the methods are called to correct their use counter.

The init-method should do initializations for the project that cannot be done in the setup-method. For example if you have to call some method defined in the project before the project can be used, this is the place to do it, because when the init-method is called all classes and methods of the project are available. This method is called every time the project wasn't in use and is used again, therefore if some action has only to be taken once then a static flag has to be used.

An example is the init-method of our demo project in section 5.6.6.4 where the method "get-editor" is called to create the user-interface. Items that should be managed by the project cannot be added to the manager with "add-inter" or "add-inter-for-opt" directly, but they have to be added to the project with

GRAPE(project, "add-item")(item)
GRAPE(project, "add-item-for-opt")(item, menu_nr)

> **PROJECT *project**
> **ITEM *item**
> **int menu_nr**

instead. These methods have the same syntax and the same effect as "add-inter" and "ad-inter-for-opt", but they allow the project to add or remove the item.

The method

GRAPE(project, "add")()

> **PROJECT *project**

is for example called when a project is added from the managers project list. First the project is "use"d in non-batch mode, then the user defined add-method, which should be called "pname-add" for project pname, is called before the project message defined by P_Version, P_Date and P_Author is displayed.

The add-method is called everytime the project is "add"ed. It can be used to create an object (or hierachy of objects) that should be included in the data hierarchy by the project. The object is declared to the project with

GRAPE(project, "set-object")(inst)

> **PROJECT *project**
> **INSTANCE *inst**

Only one object can be added to the data hierarchy (if the method is called a second time the previously given object is lost), of course you can create a whole hierarchy and give it to "set-object".

This method is used by the add-method of our demo project to add a Rot2d instance to the data hierarchy (see section 5.6.6.3).

When projects are created with g_project_include the project pointer is not available and therefore "use" and "add" cannot be called directly. Nevertheless the methods can be called with the functions

> **PROJECT *g_project_use(char *name, int batch);**
> **PROJECT *g_project_add(char *name);**

because project names are unique. The project pointer is returned by the functions to indicate that they were successful, if they return NULL something has failed, perhaps there was no project name.

5.6.5 "Unusing" or Removing a Project

The counterparts to the methods `"use"` and `"remove"` are

GRAPE(project, "unuse")()
GRAPE(project, "remove")()

PROJECT *project

As explained in the previous section `"unuse"` calls the projects exit-method, which should be called `"pname-exit"` for project `pname`, removes all classes, methods and items of the project and `"unuse"`s the projects it depends upon when the project isn't used any more. `"remove"` is called when the project is deleted from the manager's list of added projects, it just calls `"unuse"`.

The exit-method may be used to remove objects that were added by the init-method. There is no need to remove items that were given to the project by `"add-item"` or `"add-item-for-opt"`, but the *Surface* project for example adds in its init-method display methods to the display method layer provided by project *Time_Object*

```
PROJECT *project_surface_init()
{
  PROJECT *self;
  METHLAYER *layer;

  self = (PROJECT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  if(layer = (METHLAYER *)GRAPE(Time_Object, "get-display-layer")()) {
    GRAPE(layer, "add-button")("domain-disp",    "domain");
    GRAPE(layer, "add-button")("gauss-disp",     "gauss");
    GRAPE(layer, "add-button")("conjugate-disp", "conjugate");
    GRAPE(layer, "add-button")("associate-disp", "associate");
  }

  END_METHOD(self);
}
```

and it removes them from the layer in its exit-method:

```
PROJECT *project_surface_exit()
{
  PROJECT *self;
  METHLAYER *layer;

  self = (PROJECT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  if(layer = (METHLAYER *)GRAPE(Time_Object, "get-display-layer")()) {
    GRAPE(layer, "remove-button")("domain-disp",    "domain");
    GRAPE(layer, "remove-button")("gauss-disp",     "gauss");
    GRAPE(layer, "remove-button")("conjugate-disp", "conjugate");
    GRAPE(layer, "remove-button")("associate-disp", "associate");
  }

  END_METHOD(self);
}
```

You can see what happens: start a GRAPE program and add the project *Time_Object*. Then go to the option menu and open the layer by clicking on the button "geometry displays", go back to the main menu and add project *Explicit*. Now the new buttons should have appeared, they disappear again when you delete the project *Explicit*.

5.6.6 Project Rotate

The first thing we will fix is the user-interface, then we are ready to make a project — lets call it *Rotate* — out of our demo program.

5.6.6.1 A New User-Interface

As explained in section 5.5.8 most problems of the `Rot2d` interface created with the method `"get-editor"` result from binding the rulers to a fixed instance. Since we can only edit one instance at a time it would be sufficient to create one set of rulers for all `Rot2d` instances and to update the instance they point to — as the coordinate rulers variables are updated when the current point is changed.

We will place all rulers and buttons in a group to make editing easier. If the current instance is of class `Rot2d` we bind all rulers to this instance and activate the group, if no `Rot2d` instance is available we just deactivate the group so that the rulers cannot be used accidentally.

We don't have to change `"update-point"`, but we need a method to update the other rulers and to activate or deactivate the group. Since there is no special instance the method should be sent to `"update-editor"` is a class method:

```
/*
 * This time we make all rulers global. This is no problem because
 * only one interface all instances of class Rot2d is created.
 */

static GROUP *editor = NULL;
static RULER *current, *xpos, *ypos, *zpos, *angle, *discr;


CLASS *rot2d_update_editor(INSTANCE *inst)
{
  CLASS *self;
  ROT2D *rot;

  self = (CLASS *)START_METHOD(G_CLASS);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * No editor -> no update.
   */
  if(!editor)
    END_METHOD(NULL);

  /*
   * We have to make sure that the instance that was passed to this
   * method has class Rot2d (or a subclass) before we can bind the
   * rulers to it. Of course we need a curve.
   */
  if(g_has_superclass(inst, Rot2d) && (rot = (ROT2D *)inst)->curve) {
    /*
     * The method of the current ruler was already set to "update-point"
     * when it was created, now we can set the instance. Additionally
     * we limit the ruler and give it a suitable scaling.
     */
    GRAPE(current, "set-instance")(rot);
    GRAPE(current, "set-variable")(&rot->current);
    GRAPE(current, "set-min-max")
      (0.0, (double)rot->curve->number_of_points - 1.0);
    GRAPE(current, "set-scale")((double)rot->curve->number_of_points - 1.0);
```

```
    GRAPE(current, "set-offset")(0.0);

    /*
     * We may have changed the current value by hand,
     * therefore we have to update the coordinate rulers.
     */
    GRAPE(rot, "update-point")();

    /*
     * Also set the variable for angle and discretization.
     */
    GRAPE(angle, "set-variable")(&rot->angle);
    GRAPE(discr, "set-variable")(&rot->discr);

    /*
     * Finally we change the label of the group to
     * the instance's name and activate it.
     */
    GRAPE(editor, "set-label")(rot->name);
    GRAPE(editor, "set")(ACTIVE);
  } else {
    /*
     * We didn't get a Rot2d, therefore we have no instance to edit.
     * The label is set accordingly and the group is deactivated.
     */
    GRAPE(editor, "set-label")("No Object");
    GRAPE(editor, "set")(INACTIVE);
  }

  END_METHOD(self);
}
```

As you can see there are some more problems fixed now. The `current` ruler is limited, its layout is adapted to the `curve`'s number of points. The editor group gets the instance's name as label, this will help distinguishing instances if we have more than one (and they have different names).

Now here is the new `"get-editor"` method. It is a class method, too, since we create only one interface for the `Rot2d` class. It places the rulers and buttons in a group that is returned. The items are not added to the manager this time, this is left to the calling program which will add the group to the project instead of the manager. For creating the item the `new_item` function (see section 5.5.3) is used. Button and ruler sizes are relative to the group size, therefore we don't have to calculate them explicitly:

```
GROUP *rot2d_get_editor()
{
  CLASS *self;
  static int ctmp = 0, dtmp = 0.0;
  static double xtmp = 0.0, ytmp = 0.0, ztmp = 0.0;
  static double atmp = 0.0;

  /*
   * Now "get-editor" is a class method, because it only depends
   * on class Rot2d and not on a specific instance of this class.
   */
  self = (CLASS *)START_METHOD(G_CLASS);
  ASSURE(self, "", END_METHOD(NULL));

  /*
```

```
 * The editor is created once. If "get-editor" is called again,
 * only the already created editor is returned.
 */
if(!editor) {
  /*
   * We use new_item to create rulers and buttons, this allows
   * to specify methods, instances, sizes and so on with one call.
   * Because the method is called on the class we have no instance
   * to bind the rulers to, therefore we need dummy variables at
   * the moment. Later the rulers will be bound to the real instance
   * variables by "update-editor". The current ruler again gets
   * the method "update-point" to update the coordinate rulers if
   * the current point is changed. The ruler sizes are set to
   * MENU_MAX_SIZE because they are put into a group.
   */
  current = (RULER *)new_item(Ruler, I_Name, "point", I_Var, &ctmp, dfInt,
                              I_Method, "update-point", I_RSizeX, 1.0, I_End);
  xpos = (RULER *)new_item(Ruler, I_Name, "x pos", I_Var, &xtmp, dfDouble,
                           I_RSizeX, 1.0, I_End);
  ypos = (RULER *)new_item(Ruler, I_Name, "y pos", I_Var, &ytmp, dfDouble,
                           I_RSizeX, 1.0, I_End);
  zpos = (RULER *)new_item(Ruler, I_Name, "z pos", I_Var, &ztmp, dfDouble,
                           I_RSizeX, 1.0, I_End);
  /*
   * Because the angle should be in [0, 360] we limit the ruler to
   * this interval. 0 will be at the left side, 360 at the right side
   * of the ruler, it will be decremented/incremented when the arrow
   * buttons are pressed.
   */
  angle = (RULER *)new_item(Ruler, I_Name, "angle", I_Var, &atmp, dfDouble,
                            I_RSizeX, 1.0, I_MinMax, 0.0, 360.0,
                            I_Offset, 0.0, I_Scale, 360.0,
                            I_StepSize, 10.0, I_End);
  /*
   * The discretization ruler gets suitable default settings, too.
   */
  discr = (RULER *)new_item(Ruler, I_Name, "discr", I_Var, &dtmp, dfInt,
                            I_RSizeX, 1.0, I_Min, 1.0, I_Scale, 25.0,
                            I_Offset, 1.0, I_StepSize, 1.0, I_End),
  /*
   * Now we put the rulers and buttons into a group. The titel of the
   * group will be the name of the Rot2d instance we are editing.
   * The suffix "-send" was added to "insert-point" and "delete-point"
   * to avoid problems with the same methods on Triang1d.
   */
  editor = (GROUP *)
    new_item(Group, I_Size, 12.0, 8.0 + 1.25, I_Border, bfBorder|bfTitle,
             I_Item, new_item(Button, I_Method, "rotate-send",
                              I_Label, "rotate", I_RSizeX, 0.5, I_End),
             I_Item, new_item(Button, I_Method, "reset",
                              I_RSizeX, 0.5, I_End),
             I_Item, new_item(Button, I_Method, "insert-point-send",
                              I_Label, "insert-point", I_RSizeX, 0.5, I_End),
             I_Item, new_item(Button, I_Method, "delete-point-send",
                              I_Label, "delete-point", I_RSizeX, 0.5, I_End),
             I_Item, discr, I_Item, angle, I_Item, zpos,
             I_Item, ypos, I_Item, xpos, I_Item, current,
             I_End);
```

```
  }

  /*
   * We don't add the editor to the manager, this can
   * be done by the method that called "get-editor".
   */
  END_METHOD(editor);
}
```

Now all rulers have a decent layout. Unfortunately we don't have an instance to bind the rulers to, because `"get-editor"` is a class method. Rulers cannot be created without a variable therefore we use temporary variables for the moment, the rulers will be bound to a real instance by `"update-editor"` later.

5.6.6.2 The Rotate Setup-Method

Now we have everything we need to write the setup-method for our project. All method and class declarations are moved from the main program to the project file `proj_rot.c`:

```
#include <grape.h>
#include "rot2d.h"


/*
 * The class pointer for our new class
 */
CLASS *Rot2d = NULL;


/*
 * The add- and the init-function for the project, defined below.
 */
PROJECT *project_rotate_add();
PROJECT *project_rotate_init();

/*
 * Just to make life easier. We have to cast the functions pointers
 * because there is no type `pointer to an arbitrary function' in C.
 */
typedef void *(*FUNC)();


/*
 * List of all functions for our project. This list is passed to the
 * project in the "setup" method, the functions will be added to
 * the system when the project is used.
 */
static METHOD_DESCR rotate_methods[] = {
  { &Rot2d, "new-instance",      (FUNC)rot2d_new_instance     },
  { &Rot2d, "free",              (FUNC)rot2d_free             },
  { &Rot2d, "hardcopy",          (FUNC)rot2d_hardcopy         },
  { &Rot2d, "softcopy",          (FUNC)rot2d_softcopy         },
  { &Rot2d, "rotate-send",       (FUNC)rot2d_rotate_send      },
  { &Rot2d, "display",           (FUNC)rot2d_display          },
  { &Rot2d, "get-info",          (FUNC)rot2d_get_info         },
  { &Rot2d, "follow",            (FUNC)rot2d_follow           },
  { &Rot2d, "xdr",               (FUNC)rot2d_xdr              },
  { &Rot2d, "get-editor",        (FUNC)rot2d_get_editor       },
  { &Rot2d, "update-point",      (FUNC)rot2d_update_point     },
```

```
  { &Rot2d, "update-editor",      (FUNC)rot2d_update_editor     },
  { &Rot2d, "insert-point-send",  (FUNC)rot2d_insert_point_send },
  { &Rot2d, "delete-point-send",  (FUNC)rot2d_delete_point_send },

  { &Triang1d, "insert-point", (FUNC)triang1d_insert_point },
  { &Triang1d, "delete-point", (FUNC)triang1d_delete_point },

  { &Project, "rotate-add", (FUNC)project_rotate_add },
  { &Project, "rotate-init", (FUNC)project_rotate_init },

  { NULL, NULL, NULL }
};


PROJECT *project_rotate_setup()
{
  PROJECT *self;

  self = (PROJECT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * "setup" called on the project defines what classes, methods
   * and items (rulers, buttons etc) are used by the project.
   * We only add one class, therefore we don't use a class list,
   * but specify it directly. The version and date will be used
   * for the message in the manager menu.
   *
   * Note: The user-interface can not be created in this method,
   * because neither the class Rot2d is initialized nor the method
   * "get-editor" added to it. The editor is therefore added to
   * the project in the init method.
   */
  GRAPE(self, "setup")(P_Class, &Triang2d, &Rot2d, "Rot2d", sizeof(ROT2D),
                       P_MethodList, rotate_methods,
                       P_AddMethod, "rotate-add",
                       P_InitMethod, "rotate-init",
                       P_Version, 2, 0,
                       P_Date, "Aug 18, 96",
                       P_End);

  END_METHOD(self);
}
```

Since only one class is added `P_Class` is used instead of a class list. All methods were collected in a method list, their pointers have to be casted to the right type. The add- and init-method for this project were added to the list, they will be presented below.

You might have noticed that another error we made while implementing the `Rot2d` methods were corrected: the `Rot2d` method for inserting and deleting points were renamed, the missing suffix `"-send"` was added. All other methods remain unchanged.

5.6.6.3 The Rotate Add-Method

A new `Rot2d` instance should be added to the data hierarchy each time the user adds our project, therefore we provide an add-method:

```
PROJECT *project_rotate_add()
{
  PROJECT *self;
```

```
  ROT2D *rot;

  self = (PROJECT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * This method is called when a project is added from the project list
   * in the manager menu or when g_project_add is called. We just create
   * a Rot2d instance like it was done in the main program.
   */
  rot = (ROT2D *)GRAPE(Rot2d, "new-instance")("rot");
  ASSURE(rot, "project_rotate_add: can't create instance", END_METHOD(NULL));

  /*
   * We set the instance as the project's object. The project will
   * put it into the data hierarchy for us, creating a scene if
   * necessary. By added the project several time we can get any
   * number of Rot2d instances.
   */
   GRAPE(self, "set-object")(rot);

  END_METHOD(self);
}
```

5.6.6.4 The Rotate Init-Method

It was not possible to create the editor in the setup-method because the `"get-editor"` method — which is not available at that time — is needed. The editor is created in the init-method instead, a static variable is used to make sure that it is created only once:

```
PROJECT *project_rotate_init(int batch)
{
  PROJECT *self;
  static GROUP *editor = NULL;

  self = (PROJECT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * The init method is called when the project is used for the first
   * time. Here all classes are initialized and all methods added,
   * now we can create the editor (just once, therefore it has to be
   * static). The project might be called in batch mode, in this
   * case no interface is needed and we don't create it.
   */
  if(!batch && !editor) {
    /*
     * Again we need the manager, but not to add the editor to it...
     */
    MANAGER *mgr = (MANAGER *)GRAPE(Manager, "get-stdmgr")();

    /*
     * We create the editor, might fail...
     */
    ASSURE(editor = (GROUP *)GRAPE(Rot2d, "get-editor")(),
           "project_rotate_init: can't get editor", END_METHOD(NULL));

    /*
     * This time we add the editor to the project instead of the
```

```
     * manager. The project then takes care of adding the editor
     * to or removing it from the manager when the project is added
     * or removed.
     */
    GRAPE(self, "add-item")(editor);

    /*
     * We tell the manager that it should send the method "update-
     * editor" to the class Rot2d everytime its current object is
     * changed. If the new current object has class Rot2d then the
     * editor is bound to that instance, otherwise the editor is
     * deactivated.
     */
    GRAPE(mgr, "add-cross-depend")(Rot2d, "update-editor");
  }

  END_METHOD(self);
}
```

The editor is given to the project instead of the manager, it will be added and removed with the project.

You might have wondered how the `"update-editor"` method is called. Since our editor depends on the current instance of the manager we have to react when the current instance is changed. Therefore we ask the manager to include our `"update-editor"` method to its "cross-dependency list". Every time the current object is changed the manager send these methods to the given objects, in our case the class `Rot2d`, with the new current instance as parameter. This instance then is checked by `"update-editor"` and the editor is updated.

5.6.7 A Standard Main Program

All class and method declarations were removed from the main program, the `Rot2d` instance is created in `"rotate-add"` and the editor is created in `"rotate-init"` — there is almost nothing left to do for the main program. The only thing that has to be done now is to declare the new project:

```
#include <grape.h>


/*
 * All functions used in the project list have to be declared.
 */
void *project_rotate_setup();


/*
 * List of user defined programs. Each entry consists of the
 * projects name, the name of the setup method, the pointer
 * to this method and a 'hidden' flag. If the hidden flag is
 * not zero then the project will not be in the manager's
 * project list. The list has to be terminated by a null entry.
 */
static PRTYPE my_projects[] = {
  { "rotate", "rotate-setup", project_rotate_setup, 0 },

  { NULL, NULL, NULL, 0 }
};
```

```
int main()
{
  MANAGER *mgr;
  SCENE *scene;

  /*
   * Add all user defined projects to the system.
   */
  g_project_include(my_projects);

  /*
   * Get the manager and create a scene for handling.
   */
  mgr = (MANAGER *)GRAPE(Manager, "get-stdmgr")();
  scene = (SCENE *)GRAPE(Scene, "new-instance")("main");

  /*
   * Start the manager...
   */
  GRAPE(mgr, "handle")(scene);

  return 0;
}
```

A project list is used even though there is just one project at the moment, but when a new project is created is only has to be added to the list to make it available. Instead of a `Rot2d` instance a scene is created and given to the manager, the `Rot2d` instance will be created by adding the *Rotate* project from the manager's project list. Alternatively we could call

```
g_project_add("rotate");
```

before starting the manager, then the project would be added automatically.

All files for this demo project can be found in the directory `demo/rot2d/project` of the GRAPE distribution, most files are identical to the basic version presented in section 5.5.

This main program can be used as a model when projects are used for all applications. The user only needs one executable and all application are always available with a mouse click.

5.7 The GRAPE XDR Archiving System

The new GRAPE XDR archiving system provides routines to write data to and read data from files, pipes and sockets. It uses an machine-independent data format and handles references to instances and data arrays. It was written based on the experiences with the transmission of GRAPE data over sockets in the Network-Manager [OP].

The system was designed to replace the old `"read"` and `"write"` methods. All `"write"` methods used in the manager (e.g. by the `write *.st` and `write *.gr` buttons in the manag menu, see section 4.3.4) use the new routines to write the data, the corresponding `"read"` methods are able to read both the old and the new format (the `"read"` methods are still available for backward compatibility).

5.7.1 Features of the System

Writing and maintaining the new `"xdr"` methods is much easier than writing `"read"`/`"write"` methods since the new system provides many utility routines. All methods and functions are discussed in the next section where you will also find many examples (some additional examples are presented in section 5.7.6), but first we give an overview of the most important features of the GRAPE XDR archiving system:

- The data is stored in the machine-independent XDR format.

  XDR stands for "eXternal Data Representation", it essentially uses the big endian encoding for integers and the IEEE floating point format. The header files and library routines for XDR are part of Sun's Remote Procedure Call (RPC) package which is freely available and should be installed on most machines since many network services are based on it.

  Using a machine-independent data format is essential, it allows for example to write data files on an SGI and read them back in on a Linux or IBM RISC machine. This could also be achieved by using a text format instead of a binary format like XDR, but the big disadvantage of a text format is that it uses much more disk space. Currently we are working on automatic data compression using for example gzip to further reduce the data size.

  There are some memory management problems which are caused by the original XDR routines and GRAPE's own memory management, together with some minor portability problems they are discussed in section 5.7.7 where you can also find some useful hints for writing your own archiving routines.

- Data can be written to and read from files, pipes and sockets.

  This is another major advantage over the old format which could only handle files because the fseek function was used. The new format can be used on every record oriented XDR stream, this makes it possible to exchange data via pipes between GRAPE processes or even via sockets between GRAPE processes running on different machines.

- The system handles references to instances and data arrays.

  This greatly simplifies writing `"xdr"` methods since you don't have to care about writing some object more than once or resolving references when reading in data. If you know the old `"read"`/`"write"` methods for `TimeStep` compare them with the `"xdr"` method (which is presented in section 5.7.6)....

  `NULL` pointers are also handled automatically by the utility functions, this further simplifies writing `"xdr"` methods.

- Only one method is needed for both reading and writing the data.

  This is a characteristic of XDR which was adopted for all GRAPE XDR methods and functions. Using only one function should guarantee that the data written by a method or function can be read back in, it also simplifies maintaining the `"xdr"` methods and functions (for example when a data structure is changed).

5.7.2 XDR Functions

5.7.2.1 Basic XDR Functions

Before discussing the GRAPE XDR utility routines and methods we will give a very short introduction to the XDR system, for more information you should consult the XDR manual pages.

The XDR library provides a single function for each basic data type (char, short, int, long, float, double) which is used for both writing and reading. The basic form of all functions is

```
bool_t xdr_type(XDR *xdrp, type *data)
```

where `type` stands for the data type. The functions return `TRUE` if they succeed and `FALSE` if they fail.

The first parameter of all XDR functions is the XDR stream handle `xdrp`, a pointer to an `XDR` structure. This XDR stream handle can be compared to a file pointer when writing or reading a file, amongst other data it contains information about the operation that is performed: `xdrp ->x_op` is set to `XDR_ENCODE` resp. `XDR_DECODE` if the function is called for writing resp. writing.

For example the function for `int` is called `xdr_int`, it takes an `int *` as second parameter. For `unsigned int` the function is called `xdr_u_int`, this `u_` prefix has to be added for all unsigned types. The pointer to the variable is necessary because the variable value has to be changed when the function is used for reading.

The XDR library also provides routines for handling arrays and strings. These functions — besides `xdr_vector` for fixed length arrays — are dangerous because they do their own memory management, see section 5.7.7.

5.7.2.2 Basic GRAPE XDR Functions

GRAPE XDR functions are provided for all basic GRAPE types (`INT2/3/4`, `VEC2/3/4`, `FVEC3` and `MATRIX44`), they have the same format as the basic XDR functions, but they start with the prefix `g_xdr` instead of `xdr`.

For example the function for `VEC3` is

```
bool_t g_xdr_vec3(XDR *xdrp, VEC3 vec3)
{
  return xdr_vector(xdrp, (char *)vec3, 3,
                    sizeof(double), xdr_double);
}
```

In this case `vec3` already is a pointer (to a double array with three elements) therefore we don't have to pass a pointer to it to the function. `xdr_vector` is used to write/read `vec3`, it expects the XDR stream handle (`xdrp`), the base address of the array (`vec3`), the number of array elements (`3`), the size of each element (`sizeof(double)`) and the function to handle each element (`xdr_double`) as parameters.

5.7.2.3 Structures

We will use the `SUPROP_DEV` structure (see 7.1.1) which consists of four `VEC3` and two `double` variables as an example on how structures are handled:

```
bool_t g_xdr_suprop_dev(XDR *xdrp, SUPROP_DEV *sd)
{
  return (g_xdr_vec3(xdrp, sd->emission) &&
          g_xdr_vec3(xdrp, sd->ambient) &&
          g_xdr_vec3(xdrp, sd->diffuse) &&
          g_xdr_vec3(xdrp, sd->specular) &&
            xdr_double(xdrp, &sd->specular_exp) &&
            xdr_double(xdrp, &sd->transparency));
}
```

The elements of the structure are handled one after the other, combining the XDR calls with `&&` guarantees that the function is terminated and `FALSE` is returned if one of the XDR calls fails.

Currently XDR functions are provided for the GRAPE structures `REFINE2/3D`, `SUPROP_DEV`, `LIGHTSOURCE_DEV` and `DEPTHCUE_TYPE`, the name of the function always consists of the `g_xdr_` prefix followed by the structure's typedef name in lowercase letters.

5.7.2.4 Strings

Since the original XDR string function uses `malloc/free` (see section 5.7.7 on this problem) we provide a function with the same functionality which uses the GRAPE memory management functions instead:

```
bool_t g_xdr_string(XDR *xdrp, char **str)
```

Again a pointer to the string (`char` array) is needed as second parameter.

You should always use this function which also supports `NULL` strings to write strings, for an example see the `Scene "xdr"` method below.

Using strings might lead to machine-dependent data, see section 5.7.7 for a discussion of this problem.

5.7.2.5 Arrays

There are two functions in the XDR library for handling arrays: `xdr_array` for arbitrary and `xdr_vector` for fixed length arrays. You can use `xdr_vector` (see the `g_xdr_vec3` function above) but you should not use `xdr_array` because it uses `malloc/free` (like the XDR string function, again consider section 5.7.7).

The GRAPE function for arrays provides much more functionality than the original XDR functions, its syntax is

```
bool_t g_xdr_array(XDR *xdrp, void **array, int number_of_elements,
                   int max_number_of_elements, char *typename)
```

Like for the other functions a pointer to the array pointer is the second parameter, the third and fourth parameter are the number of elements and the maximal number of elements in the array.

The last parameter is a string which contains the name of the type of the array, e.g. `"int"` for an integer array or `"unsigned long"` for an unsigned long array. The basic C and GRAPE types, `"char *"` for string arrays and `"INSTANCE *"` for instance arrays are predefined, you can add you own types with the function

```
int g_xdr_add_type(char *name, int size, xdrproc_t xdrfunc)
```

which has the type's name (`"type"`), it's size (`sizeof(type)`) and the XDR function which handles variables of this type (`xdr_type` or `g_xdr_type`) as parameters.

The `g_xdr_array` function handles arrays as instances of a special class `Array` (which is for internal use only), it is responsible for handling

- references,
- `NULL` pointers and
- the memory management.

Handling references is the most important feature. For example if you work with time-dependent data on a fixed triangulation (i.e. many `Triang2d` or subclass instances with the same vertex/neighbour indices) you can use the same `vertex` and `neighbour` arrays for all instances (though this might be dangerous, e. g. when data is shared between instances). If you write these instances `g_xdr_array` is called several times on the same arrays (see section 5.7.6 for the `Triang2d` `"xdr"` method). On the first call the array itself is written, on all subsequent calls only a reference is written. When the data is read back in and `g_xdr_array` finds a reference it just returns the pointer to the already read array — the real array is always stored ahead of any reference to it.

If on reading the maximal number of elements of the parameter array differs from the maximal number of elements read the given array will be freed and new memory allocated. Therefore if the given array parameter is not `NULL` (the pointer, not the pointer pointer!) it should have been allocated with `mem_alloc` or another memory function like `int_alloc` which internally uses `mem_alloc`.

5.7.3 XDR Methods

Basically instances are handled like structures, but instead of a function a method called `"xdr"` is used. `"xdr"` methods exist for all geometry and hierarchy classes, for a complete list see the index.

The `Scene` `"xdr"` method is an example how the basic functions can be used to write/read an instance:

```
SCENE *scene_xdr(XDR *xdrp)
{
  SCENE *self;
  int ver = 1;
```

```
    self = (SCENE *)START_METHOD(G_INSTANCE);
    ASSURE(self, "", END_METHOD(NULL));

    if(!g_xdr_version(xdrp, &ver) ||
       !g_xdr_instance(xdrp, (INSTANCE **)&self->next_scene) ||
       !g_xdr_instance(xdrp, (INSTANCE **)&self->object) ||
       !g_xdr_matrix44(xdrp, self->object_trans) ||
       !  xdr_int(xdrp, &self->matrix_flag) ||
       !g_xdr_string(xdrp, &self->method_name) ||
       !g_xdr_instance(xdrp, (INSTANCE **)&self->suprop))
      END_METHOD(NULL);

    END_METHOD(self);
  }
```

You should ignore the `g_xdr_version` function at the moment, version control is explained in section 5.7.4.

The matrix information is handled with `g_xdr_matrix44` and `xdr_int`, the `Scene` method name with `g_xdr_string`. A `NULL` `self->method_name` pointer is handled automatically by the `g_xdr_string` function, if necessary memory for `self->method_name` is allocated on reading.

Again the XDR calls are combined, this time with `!` and `||` which has (almost) the same effect as the `&&` in the structure example above.

5.7.3.1 g_xdr_instance

The most important function used in this example is the function for writing and reading instances:

```
bool_t g_xdr_instance(XDR *xdrp, INSTANCE **inst)
```

which takes an pointer to an instance pointer as second parameter. It is responsible for handling (compare this with the `g_xdr_array` function in the previous section)

- references,
- `NULL` pointers,
- the memory management and
- skipping unknown instances.

References are handled in the same way as for arrays: the first time `g_xdr_instance` is called on an instance it is written/read, all subsequent calls on this instance just create/resolve references. When a reference is resolved while reading, the `refcount` of the instance is corrected (which is not possible for arrays).

Memory management is only necessary in two cases when the function is called for reading: if the given instance is `NULL` (again the pointer, not the pointer pointer!) or has the wrong class a new instance is created, in the second case the given instance is deleted.

Skipping unknown data can be very useful. If you create for example a subclass of `Adapt2d` with its own `"xdr"` method for your numerics and write an instance of this class to a file, somebody else who hasn't got this subclass and its `"xdr"` method will still be able to read the file — in this case an `Adapt2d` instance will be created and the subclass data is ignored — and to visualize the data.

Additionally `g_xdr_instance` writes/reads some class information before calling the appropriate `"xdr"` method. Therefore:

NEVER write an instance directly using the `"xdr"` method, but ALWAYS use `g_xdr_instance` instead.

5.7.3.2 g_xdr_superclass

The same rule applies to calling "^xdr" (i.e. calling the "xdr" method on the superclass), always use the function

```
bool_t g_xdr_superclass(XDR *xdrp, INSTANCE *inst)
```

instead. Since `inst` always exists at the point where `g_xdr_superclass` is called only an instance pointer is needed as second parameter.

Like `g_xdr_instance` this function is responsible for handling class information, especially skipping unknown (super)classes, and calling "xdr" on the superclass.

As example we use the "xdr" method on the `Chain` subclass of `Scene` which just adds a flag:

```
CHAIN *chain_xdr(XDR *xdrp)
{
  CHAIN *self;
  int ver = 1;

  self = (CHAIN *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  if(!g_xdr_version(xdrp, &ver) ||
     !g_xdr_superclass(xdrp, (INSTANCE *)self) ||
     !  xdr_int(xdrp, &self->flag))
    END_METHOD(NULL);

  END_METHOD(self);
}
```

`g_xdr_superclass` should always be called right after `g_xdr_version` and before any other XDR functions on the instance variables (this is not really necessary but it keeps the structure of all "xdr" methods consistent).

5.7.4 Version Control

If a class or structure is changed the "xdr" method or `g_xdr_` function for the new version should still be able to read old data files. The function

```
bool_t g_xdr_version(XDR *xdrp, int *ver)
```

provides a simple version control mechanism.

When the function is called for writing the current version number (1 in the `Scene` and `Chain` examples above) is written. If the function is called for reading it reads in the version and compares it to the current version which was given as the second parameter. If the current version is lower then the version number read the function fails and returns `FALSE` — in this case somebody else has written the data with a newer version of the method/function and the current version is not able to handle the data. Therefore

ALWAYS call `g_xdr_version` before any other XDR function

or you get into trouble when you have to change the class or structure.

Look at the "xdr" method on class `Suprop` which is called in the `Scene` "xdr" method above via the `g_xdr_instance` call on `&self->suprop`:

```
SUPROP *suprop_xdr(XDR *xdrp)
{
  SUPROP *self;
  int ver = 2;
  int grid_patch, enabled;
```

```
  self = (SUPROP *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  grid_patch = (self->grid_patch == G_GRID ? 1 : 0);
  enabled = (int)self->enabled; /* new version */

  if(!g_xdr_version(xdrp, &ver) ||
     !g_xdr_suprop_dev(xdrp, &self->suprop_dev) ||
     !  xdr_int(xdrp, &grid_patch))
    END_METHOD(NULL);
  /* new version -> */
  if(ver == 1) {
    g_vec3_set(self->linecolor, 1.0, 1.0, 1.0);
    enabled = (int)spefDefault;
  } else
    if(!  xdr_int(xdrp, &enabled) ||
       !g_xdr_vec3(xdrp, self->linecolor))
      END_METHOD(NULL);
  /* <- new version */

  self->grid_patch = (grid_patch ? G_GRID : G_PATCH);
  self->enabled = (SUPROP_ENABLE_FLAG)enabled; /* new version */

  END_METHOD(self);
}
```

The old version of class `Suprop` just contained the `suprop_dev` and `grid_patch` variables, later the `linecolor` and an `enabled` flag were added, therefore the lines marked with the "new version" comment were added to the method.

If the method is used for writing or for reading a new version the `else` part is used. If an old version is read the `if` part is used, in this case the new variables have to be initialized to default values.

The `enabled` variable in copied to a local `int` variable (and back, this has no effect if the method is called for writing) because it is an `enum` and there is no special function for handling enums. The `grid_patch` flag is converted from `G_GRID`/`G_PATCH` to 1/0 (and back, again this has no effect on writing) because these defines were changed several times, otherwise for each change a new version of this routine would have been required.

5.7.5 XDR Streams

Since the functions described in the previous sections all work with any XDR stream they can be used on files, pipes and sockets. Don't open more than one XDR stream at a time (see section 5.7.7), this problem will hopefully be fixed in the next library version.

5.7.5.1 Files

Writing or reading files is supported with two functions:

```
XDR *g_xdr_open_file(char *filename, OPEN_MODE mode)
```

works like `fopen`, it opens the file `filename` for writing (if `mode` is `omWrite`) or reading (if `mode` is `omRead`) and returns the pointer to the XDR stream created for this file. A file created by `g_xdr_open_file` is closed by

```
int g_xdr_close_file(XDR *xdrp)
```

The functions return NULL resp. FALSE if they fail.

With these functions writing an GRAPE XDR file is as simple as writing a normal file. The code to write the instance inst to the file filename is (to read the instance just replace omWrite by omRead)

```
if((xdrp = g_xdr_open_file(filename, omWrite))) {
  if(!g_xdr_instance(xdrp, &inst))
    fprintf(stderr, "error writing instance\n");
  g_xdr_close_file(xdrp);
} else
  fprintf(stderr, "error opening file\n");
```

Of course you can write any kind of data to the file, you just have to replace the g_xdr_instance function by the appropriate XDR functions.

5.7.5.2 Pipes and Sockets

If you are not familiar with pipes or sockets and XDR programming you should skip this section or consider the manual pages first.

There is no special support to create pipes or sockets in GRAPE. But if you have a pipe or socket, transmitting data over it is as simple as writing to or reading from a file. First you have to create an XDR stream by calling xdrrec_create and to set the XDR stream mode xdrp->x_op to XDR_ENCODE for writing or XDR_DECODE for reading, for reading you additionally have to call xdrrec_skiprecord. Now you can write or read the instance inst with

```
  if(!g_xdr_start(xdrp) ||
     !g_xdr_instance(xdrp, &inst) ||
     !g_xdr_end())
    fprintf(stderr, "transmission failed\n");
```

On writing you now have to call xdrrec_endofrecord to flush the data, finally the XDR stream can be destroyed with xdr_destroy.

The g_xdr_start functions writes/reads a magic number — "GXDR" in bytes 9–12 of the stream — and the global version and revision number (which can be obtained by g_xdr_get_version and g_xdr_get_revision and is for internal use only). g_xdr_end is needed to clean up some internal variables after the data has been transmitted.

For files g_xdr_open_file opens the file, creates the stream and writes the header, and g_xdr_close_file destroys the stream, cleans up and closes the file.

5.7.6 More Examples

5.7.6.1 Rot2d "xdr"

With the utility functions presented in the last section writing an "xdr" method for our class Rot2d from section 5.3.5 is no problem:

```
ROT2D *rot2d_xdr(XDR *xdrp)
{
  ROT2D *self;
  int ver = 1;

  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  if(!g_xdr_version(xdrp, &ver) ||
     !g_xdr_superclass(xdrp, (INSTANCE *)self) ||
     !g_xdr_instance(xdrp, (INSTANCE **)&self->curve) ||
```

```
    !g_xdr_instance(xdrp, (INSTANCE **)&self->axis) ||
    !  xdr_double(xdrp, &self->angle) ||
    !  xdr_int(xdrp, &self->discr) ||
    !  xdr_int(xdrp, &self->current))
   END_METHOD(NULL);

  END_METHOD(self);
}
```

We just write version, the superclass and all instance variables. Of course we have to add the `"xdr"` method to the project's method list before we can use it.

Now you can try out what happens if an `"xdr"` method or a class is missing. Start the program, add the *rotate* project, create a rotational surface and write it to a file using the `write *.gr` button in the manag menu. Now exit the program, start it again and read the file back in using the `read *.gr` button without adding the *rotate* project. You get just a `Triang2d` instance without axis and curve, since the `Rot2d` class and its `"xdr"` method were missing the data for this class was skipped and only the superclass data was read.

5.7.6.2 TimeStep "xdr"

This is a very good example on how references and `NULL` pointers are handled. The `pre_object` and `post_object` of a `TimeStep` instance can be identical, the `pre_step` or `post_step` can be `NULL`, the steps even can be cyclic (first pointing to last and vice versa). Nevertheless the `TimeStep` `"xdr"` method is very simple:

```
TIMESTEP *timestep_xdr(XDR *xdrp)
{
  TIMESTEP *self;
  int ver = 1;

  self = (TIMESTEP *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  if(!g_xdr_version(xdrp, &ver) ||
     !  xdr_double(xdrp, &self->time) ||
     !g_xdr_instance(xdrp, (INSTANCE **)&self->pre_step) ||
     !g_xdr_instance(xdrp, (INSTANCE **)&self->post_step) ||
     !g_xdr_instance(xdrp, (INSTANCE **)&self->pre_object) ||
     !g_xdr_instance(xdrp, (INSTANCE **)&self->post_object))
    END_METHOD(NULL);

  END_METHOD(self);
}
```

Compare this to the 140 lines of code of the old `"read"`/`"write"` methods which had to do all these checks for `NULL` pointers and references. Still the old routines were not able to detect all references correctly, i.e. if some objects were referenced somewhere outside the `TimeStep` sequence these references couldn't be detected.

5.7.6.3 Triang2d "xdr"

This is a rather long example, but it shouldn't be too difficult to understand. It shows how arrays and memory management can be handled:

```
TRIANG2D *triang2d_xdr(XDR *xdrp)
{
  TRIANG2D *self;
  int number_of_points;
```

```
  int max_number_of_points;
  int number_of_elements;
  int max_number_of_elements;
  int ver = 1;

  self = (TRIANG2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /* for reading -> */
  number_of_points = self->number_of_points;
  max_number_of_points = self->max_number_of_points;
  number_of_elements = self->number_of_elements;
  max_number_of_elements = self->max_number_of_elements;
  /* <- for reading */

  if(!g_xdr_version(xdrp, &ver) ||
     !  xdr_int(xdrp, &number_of_points) ||
     !  xdr_int(xdrp, &max_number_of_points) ||
     !  xdr_int(xdrp, &number_of_elements) ||
     !  xdr_int(xdrp, &max_number_of_elements))
    END_METHOD(NULL);

  /* for reading -> */
  if(max_number_of_points != self->max_number_of_points ||
     max_number_of_elements != self->max_number_of_elements) {
    GRAPE(self, "list-free")();
    self->max_number_of_points = max_number_of_points;
    self->max_number_of_elements = max_number_of_elements;
  }
  self->number_of_points = number_of_points;
  self->number_of_elements = number_of_elements;
  /* <- for reading */

  if(!g_xdr_array(xdrp, (void **)&self->x, self->number_of_points,
                  self->max_number_of_points, "double") ||
     !g_xdr_array(xdrp, (void **)&self->y, self->number_of_points,
                  self->max_number_of_points, "double") ||
     !g_xdr_array(xdrp, (void **)&self->z, self->number_of_points,
                  self->max_number_of_points, "double") ||
     !g_xdr_array(xdrp, (void **)&self->vertex, self->number_of_elements,
                  self->max_number_of_elements, "INT3") ||
     !g_xdr_array(xdrp, (void **)&self->neighbour, self->number_of_elements,
                  self->max_number_of_elements, "INT3"))
    END_METHOD(NULL);

  END_METHOD(self);
}
```

The `Triang1d` and `Triang3d` `"xdr"` methods are very similar to this one, for `Triang3d` `INT3` has to be replaced by `INT4` and for `Triang1d` all lines referring to element information have to be removed.

The two parts marked with the "for reading" comments are not needed for writing, in this case the `number_of_` variables are just copied to local variables and back again, the `if` part with the `"list-free"` is not executed. The arrays might be `NULL` (if `max_number_of_points/elements` is zero), they might also be used somewhere else, e.g. in some other `Triang2d` instance, this is all handled by `g_xdr_array`.

If the method is called for reading, first the (maximum) number of points and elements are read into local variables. Reading them directly into the corresponding `Triang2d` variables would cause lots of trouble because the `max_number_of_` variables probably wouldn't be consistent with the array lengths. If the existing arrays don't have the correct size we delete them by calling `"list-free"`, then the local variables can be copied to the `Triang2d` variables. We could call `"list-alloc"` to allocate new arrays, but we don't have to since `g_xdr_array` will allocate new memory if necessary.

With some minor changes the method would work even without calling `"list-free"` at all, in this case the array lengths would be corrected by `g_xdr_array` if necessary. But this could also lead to inconsistencies if the method was called on a subclass instance: arrays in the subclass which contain point or element data would still have the old lengths but these old lengths are lost when the method returns. Calling `"list-free"` also deletes subclass arrays, new arrays with the correct lengths are then allocated by the `g_xdr_array` calls in the subclass `"xdr"` method.

5.7.7 Problems

In this section we will discuss the problems mentioned in the previous sections and give some hints for writing XDR methods and functions.

5.7.7.1 Memory Management

The XDR library functions don't know anything about GRAPE and its own memory management, they use `malloc` and `free` to allocate and free memory. You should be careful not to allocate memory for a GRAPE instance with `malloc` because it would be freed with `mem_free` when the instance is deleted. This would lead to an error message from the GRAPE memory manager, accidentally freeing memory allocated with `mem_alloc` with `free` however would crash the program. The safest way to avoid these problems is to not use XDR functions which allocate or free memory, instead of `xdr_string` and `xdr_array` you should use the corresponding GRAPE XDR functions `g_xdr_string` and `g_xdr_array`.

Another problem is reading strings or arrays into static or local arrays which are not dynamically allocated. If an XDR function tries to free such an array which is located in the program's data segment or on the stack the program will probably crash. Since static or local strings (char arrays) are very common, `g_xdr_string` never frees the array if the string to be read doesn't fit into it but just allocates new memory. If the string was allocated dynamically this data block will be lost. As described in section 5.7.2 `g_xdr_array` behaves differently: because very large memory blocks could be lost, the old data is freed if necessary before new memory is allocated, therefore the parameter array should have been allocated dynamically.

5.7.7.2 Strings and Opaque Data

Strings are written as opaque data, i.e. in their binary representation, therefore the result depends on the machines character set. Using just the basic characters a-z, A-Z, 0-9 should be save because their representation is identical in most character sets (they could cause trouble when a 16 bit character set is used), but language specific characters will lead to unexpected result.

To create machine-independent files (apart from strings) it is necessary to write all data with the appropriate XDR functions. For opaque data pointers like the `char * data` pointer in class `Fe2d` it is impossible to determine which XDR function has to be used, therefore such data can only be written in its machine-dependent binary representation. To avoid this problem the `Fe2d` `data` is only written if its size `size_of_data` is a multiple of `sizeof(double)`. In this case `data` is expected to point an array of `double`s and the data is written using `xdr_double` — if `data` points to some other data type the resulting data will be machine-dependent.

5.7.7.3 Function Pointers

Archiving functions pointers obviously is not possible. The `Fe2d` function pointer `f` for example is ignored, it has to be set to the correct function when data is read in. There are two ways to solve this problem: using subclasses or indices.

Instead of using the `Fe2d data` pointer to store the FE data you should create a subclass with its own data pointer which has a specific type. The subclass' `"xdr"` function then can write/read the data without problems, it also can restore the correct FE function on reading.

Another possibility is to assign an index to each function. On writing the index for the function is written, on reading the function can be restored from the index. Of course this is only possible if a fixed set of functions is used.

5.7.7.4 `G_List` and `GraphicDevice`

A `G_List` can only be written if the type of its elements can be determined, therefore you first have to set the type with `"set-type"`. `G_List` uses the same types as `g_xdr_array`, therefore new types can be added with `g_xdr_add_type` if necessary, see section 5.7.2.

There is no `"xdr"` method for class `GraphicDevice`, instead the function `g_xdr_graphicdevice` has to be used. The reason for this exception is that it is very dangerous to read in `GraphicDevice` instances with `g_xdr_instance`, because new `GraphicDevice` instances could be created or — even worse — existing instances could be deleted.

5.7.7.5 `char` and `short` Arrays

You should avoid using `xdr_short` and especially `xdr_char` since they write their argument as `int`s and therefore waste disk space or network resources. The `REFINE2/3D` structures are for example packed into two resp. three `int`s on writing and unpacked on reading.

5.7.7.6 Limitations of the Implementation

In the current implementation the GRAPE XDR functions uses a global hash table to store references, therefore it is not possible to use more than one XDR stream at a time. If you try this a stream might contain references to an object with is only contained in another stream, on reading this will result in errors.

Data compression is experimental at the moment (using an external program like gzip), it will probably be available in the next library version. We still consider using an internal compression since using fork and pipes will not work on all platforms.

5.7.7.7 Compile or Link Problems

If the XDR header files are missing you will not be able to compile GRAPE programs, in this case you should get the RPC package which is freely available from Sun and install it on the system.

The XDR library routines might not be contained in the C library, in this case you probably have to link your program with the sun library (-lsun).

Please consider the XDR man pages or ask the system operator if you have one of these problems.

Chapter 6

Geometry Classes

6.1 Points

We have no special class Point. Since this would cause too much overhead we suggest the usage of the structure `VEC2` or `FVEC2` in two space, `VEC3` or `FVEC3` in three space and `VEC4` or `FVEC4` in four space when working with single points. Be aware that these structures are no **GRAPE** classes, they are defined as

```
typedef FVEC2 float[2];
typedef VEC2 double[2];
typedef FVEC3 float[3];
typedef VEC3 double[3];
typedef FVEC4 float[4];
typedef VEC4 double[4];
```

The float types are no longer supported by any function or method, they just exist.

Clouds of points should be stored in the coordinate list of structures of class `Triang1d`, `Triang2d` or `Triang3d`. For this purpose the method `"point-list-alloc"` allocates only the coordinate lists and ignores the connectivity. Using the display method `"points-disp"` one can visualize the points as little spheres with variable radius. Surely this method also works on regular triangulations.

`VEC2`, `VEC3` and `VEC4` are supported by a huge number of utility functions, which are listed in section 10.2.

The xdr functions for `VEC2`, `VEC3` and `VEC4` are `g_xdr_vec2`, `g_xdr_vec3` and `g_xdr_vec4`, respectively (see 5.7).

6.2 Curves

The simplest classes for geometries in **GRAPE** are for curves. The main class for curves is `Triang1d`, its entries consist of a sequence of points in three space. For this class a number of display methods exist which e.g. show the data as ordinary line or as a tube.

The subclass `Bnd_Curve` is used for curves whose points have a direct connection with the surface, e.g. for boundary curves of a surface patch. Still these curves can exist on their own because they have their own coordinate arrays. Curves of class `Cons` don't have their own coordinate arrays, therefore they always belong to a geometry.

Further curve representations are the classes `Bezier1d` and `Spline1d`. Currently they are only used by the `FunctionRuler` 9.1.3.10 and the `SplineEditor` 4.1.2.10.

Additionally to these static data structures there exist time dependent curves of class `Time_Curve`, they are described in sections 7.4.2 and 8.2.2.

6.2.1 Triang1d

The class `Triang1d` contains purely vertex information:

```
#CLASS Triang1d : Root {
  int number_of_points;       /* currently used number of points */
  int max_number_of_points;   /* length of list vectors */
  double *x, *y, *z;          /* point coordinates */
};
```

6.2.1.1 Memory Methods on `Triang1d`

GRAPE(triang1d, "free")()

TRIANG1D *triang1d

The associated memory of the instance variables is freed up and the instance is deleted. For a more detailed description of this method and its usage consult the description of the `"delete"` method (5.4.4.1).

GRAPE(triang1d, "list-alloc")(max_points)

GRAPE(triang1d, "list-growto")(max_points)

GRAPE(triang1d, "list-free")()

TRIANG1D *triang1d
int max_points

`"list-alloc"` allocates or resizes memory for `max_points` points for the coordinate lists. `"list-growto"` assures that at least `max_points` points are available, by calling `"list-alloc"` only if `triang1d->max_number_of_points` ¡ `max_points`. `"list-free"` frees the memory for the coordinate lists and sets `triang1d->max_number_of_points` to 0. For a description of the general concept see 5.4.4.2.

GRAPE(triang1d, "list-shrink-send")()

TRIANG1D *triang1d

Free unused space of the vertex lists: resize the vertex lists to hold exactly `triang1d ->number_of_points` points. This is done by calling `"list-alloc"` with this value as `max_points` argument.

This method is called by

GRAPE(triang1d, "optimize-mem-send")()

TRIANG1D *triang1d

Free unused space of the vertex lists.

GRAPE(triang1d, "hardcopy")()

TRIANG1D *triang1d

Create an identical copy of `triang1d` which has its own lists (all data is copied). The pointer to the new instance is returned.

GRAPE(triang1d, "softcopy")(t1_copy)

TRIANG1D *triang1d, *t1_copy

Make a softcopy of a `Triang1d` instance: like `"hardcopy"` but only the used data (up to `number_of_points`) is copied. If `t1_copy` is not `NULL` then this instance is used to store the copy otherwise a new instance is created.

(int *)GRAPE(t1_one, "merge")(t1_two)

TRIANG1D *t1_one, *t1_two

Merge two `Triang1d` instances: append the vertex lists of `t1_two` to those of `t1_one`. `t1_two` is left untouched. If the method fails, it returns `NULL`. Both `z` vectors of `t1_one` and `t1_two` may be `NULL`, but only at the same time.

This method is designed to merge curves, it requires the last vertex of `t1_one` to be equal to the first vertex of `t1_two`.

6.2.1.2 Display Methods on `Triang1d`

GRAPE(triang1d, "display")()

TRIANG1D *triang1d

The instance `tr` is rendered on the standard device in a default manner. In fact `"display"` is an interactive interface for `"pipe"` (see below).

GRAPE(triang1d, "pipe")(top, shape, tail, scal, diskr, eps)

TRIANG1D *triang
void (*top)(arg,x,y)
/* function describing the top, arg between 0 and 1 */
void (*tail)(arg,x,y)
/* describing the end, arg between 0 and 1 */
void (*shape)(arg,x,y)
/* function describing cross-section, arg between -pi/2 and pi/2 */
double scal
/* factor for scaling */
int diskr
/* parameter for discretization */
double eps
/* parameter to change between the */
/* calculation with the Frenet frame */
/* or a simpler one */

A pipe around a `Triang1d` instance is constructed and displayed on the standard device. The shape of the spike, the end and the appearance of a cross–section of the pipe can be specified by function descriptions. Default functions for `top`, `tail` and `shape` are function descriptions of an arrow, an ellipse and a ball (arrow, ellipse, ball). The function description of a ball is used if one of the function variable is a `NULL` pointer.

The number of polygons which are put together to build the pipe can be varied with the parameter `diskr`. The size of the cross–section can be manipulated with `scal`. With a small value `eps` (default) the calculation of the pipe is made with the Frenet frame. A very small value of the second derivative of the curve determined through the polygon causes problems in the calculation of the pipe. Therefore the possibility to work not with the Frenet frame is given through the variable `eps` representing a lower bound for the second derivative.

GRAPE(triang1d, "tube-disp")()

TRIANG1D *triang1d

The instance `triang1d` is rendered on the standard device in a default manner. `"tube-disp"` calls `g_draw_tube` (which is a faster and less flexible version of `"pipe"`) to draw a tube around the `Triang1d` instance.

GRAPE(triang1d, "points-disp")()

GRAPE(triang1d, "points")(radius)

TRIANG1D *triang1d
double radius

The `Triang1d` instance is rendered as a cloud of points. The ball `radius` is controlled by a ruler which is added to the manager by the interactive method `"points-disp"`.

6.2.1.3 I/O Methods on `Triang1d`

(TRIANG1D *) GRAPE(Triang1d, "read")()

CLASS *Triang1d

Read a `Triang1d` instance from the standard textdevice. If successful the instance otherwise `NULL` is returned. This method is only provided for backward compatibility to read files written with `"write"`, it is replaced by

GRAPE(triang, "xdr")(xdrp)

TRIANG1D *triang
XDR *xdrp

Read the instance `triang` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

6.2.1.4 Other Methods on `Triang1d`

GRAPE(triang, "interpol")(tr1, tr2, factor)
GRAPE(triang, "interpol-fac")(tr1, tr2, fac1, fac2)

TRIANG1D *triang, *tr1, *tr2
double factor, fac1, fac2

Linear interpolation of the two `Triang1d` instances `tr1` and `tr2` with `factor`:

```
triang = (1 - factor) * tr1 + factor * tr2
```

or with factors `fac1` and `fac2`:

```
triang = fac1 * tr1 + fac2 * tr2.
```

If necessary new vertex lists are allocated for `triang`.

The two `Triang1d` instances are interpolated pointwise, therefore they must have the same `number_of_points`.

GRAPE(triang, "reflect")(trfmat, orient)

TRIANG1D *triang
MATRIX44 trfmat
int orient

Create a new `Triang1d` instance from `triang` with `triang->number_of_points` points which are transformed with `trfmat`. If `trfmat` is `NULL` the points are copied without transformation. The orientation flag `orient` is not needed but supplied so that all reflection methods have identical parameter lists. The reflected `Triang1d` instance is returned.

GRAPE(triang, "model")(mat)

TRIANG1D *triang
MATRIX44 mat

Transform all vertices of `triang` with `mat`.

GRAPE(triang1d, "close-send")()

TRIANG1D *triang1d

Overwrite the last point of `triang1d` with the first one to assure it is closed.

GRAPE(triang1d, "refine-global-send")()

TRIANG1D *triang1d

Insert a new vertex in the middle of each two vertices of `triang1d`. A non existing `triang1d->z` is handled correctly.

GRAPE(triang, "get-bnd-box")(xmin, xmax, ymin, ymax, zmin, zmax)

TRIANG1D *triang
double *xmin, *xmax, *ymin, *ymax, *zmin, *zmax

Calculate the bounding box of `triang`, i.e. fill the parameters with the minima and maxima of the vertex values.

There exists a method `"insert-bnd-box-send"`() on class `Root` which inserts a bounding box into the data hierarchy tree, only requiring a `"get-bnd-box"`-method on the class.

GRAPE(triang1d, "get-info")(list)

TRIANG1D *triang1d
LIST *list

Create info `list` entries for a `Triang1d` instance. Entries are class, name, number of points and maximum number of points.

GRAPE(triang1d, "show-send")()

TRIANG1D *triang1d

Print the data of a `Triang1d` instance to `stdout`.

GRAPE(tr, "c-show")()

TRIANG1D *tr

The data of an instance `tr` is written to the standard output. The output looks like a **C** source file.

GRAPE(triang1d, "reset")()

TRIANG1D *triang1d

Reset `triang1d->number_of_points` to zero (only needed in universal mode (see 5.4.8)).

6.2.2 Bnd_Curve

The class `Bnd_Curve` adds instance variables to `Triang1d` which allow points of the curve to lie on an associated surface. The surface is connected to the `Bnd_Curve` via the `Triang2d` pointer `geom`, the `i`th point of the `Bnd_Curve` then corresponds to the `vertex[i]`th point of `geom`.

Since `Bnd_Curve` instances have their own coordinate arrays they don't depend entirely on the surface, most methods will also work with a `NULL` `geom` pointer. Changing the surface `geom` will not change the `Bnd_Curve` instance directly, this is done by calling `"assign-curve-send"`.

Instances of this class are used as boundary curves of `Geom2d` instances. In this case the `neighbour` variable contains the indices of the triangles along the `Bnd_Curve`, this allows to easily create the new connectivity if two `Geom2d` are merged and closed. Additionally information about the type of the curve (planar, circular etc.) can be stored in the `restrict` structure.

```
#CLASS Bnd_Curve : Triang1d {
  int flag;       /* flag if bnd_curve ACTIVE */
  TRIANG2D *geom; /* pointer to triang2d to which the bnd_curve belongs */
                  /* only to be changed with "set-geom" */
  int *vertex;    /* index of corresponding point in triang2d */
  int *neighbour; /* index of triangle which the next edge belongs to */
  struct {                          /* restriction information for curve */
      void (*f)(struct bnd_curve *); /* subject to changes in the future */
      int size;
      double *data;
  } restrict;
};
```

6.2.2.1 Memory Methods on Bnd_Curve

(BND_CURVE *) GRAPE(Bnd_Curve, "new-instance")(name)

CLASS *Bnd_Curve
char *name

Create a `Bnd_Curve` instance with name `name`. Only the structure itself is allocated and the `flag` is set to `ACTIVE`. The `vertex` and `neighbour` lists have to be allocated separately using `"list-alloc"`.

GRAPE(bnd_curve, "set-geom")(triang)

BND_CURVE *bnd_curve
TRIANG2D *triang

This method assigns a surface to `bnd_curve`. The `geom` pointer should only be accessed by this method which also takes care of the refcounting: an existing `geom` instance is deleted first, then the new one is `ASSIGN`ed.

GRAPE(bnd_curve, "free")()

BND_CURVE *bnd_curve

Free a `Bnd_Curve` instance, see section 5.4.4.1.

GRAPE(bnd_curve, "list-alloc")(length)
GRAPE(bnd_curve, "list-free")()

BND_CURVE *bnd_curve
int length

`"list-alloc"` allocates memory for `length` points for the vertex and neighbour lists, `"list-free"` frees these lists, see section 5.4.4.2.

GRAPE(bnd_curve, "delete-restrict")()

BND_CURVE *bnd_curve

Remove a restriction and free the associated memory.

GRAPE(bnd_curve, "list-shrink-send")()
GRAPE(bnd_curve, "optimize-mem-send")()

BND_CURVE *bnd_curve

`"list-shrink-send"` frees unused space of the `bnd_curve` lists by calling `"list-alloc"` with `number_of_points` points as argument. It is called by `"optimize-mem-send"` which only is provided for compatibility.

GRAPE(bnd_curve, "hardcopy")()
GRAPE(bnd_curve, "softcopy")(copy)

BND_CURVE *bnd_curve, *copy

Create a copy of `bnd_curve` which its own vertex lists (all data is copied) and return the pointer to it. The `geom` instance is only `ASSIGN`ed to the copy, therefore the copy is associated with the same `Triang2d` instance as the original `bnd_curve`.

`"hardcopy"` and `"softcopy"` differ in two respects. While `"hardcopy"` creates an identical copy of `bnd_curve`, `"softcopy"` only copies the memory needed, i.e. upto `number_of_points`. If `copy` is `NULL`, `"softcopy"` creates a new `Bnd_Curve` instance to store the copy, otherwise `copy` is used.

6.2.2.2 Display Methods on `Bnd_Curve`

GRAPE(bnd_curve, "display")()

BND_CURVE *bnd_curve

Display a `Bnd_Curve` instance using the same `"pipe"` method as for `Triang1d`. This method is only supplied to allow separate rulers for `Bnd_Curve` and `Triang1d` instances.

6.2.2.3 I/O Methods on `Bnd_Curve`

GRAPE(bnd_curve, "xdr")(xdrp)

BND_CURVE *curve
XDR *xdrp

Read the instance `bnd_curve` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

6.2.2.4 Debug Methods on `Bnd_Curve`

GRAPE(bnd_curve, "show-send")()

BND_CURVE *bnd_curve

Print the data of a `Bnd_Curve` instance to `stdout`.

6.2.2.5 Other Methods on `Bnd_Curve`

GRAPE(bnd_curve, "assign-curve-send")()

BND_CURVE *bnd_curve

Assign the `geom->x/y/z` values for the `bnd_curve->vertex` indices from the associated `Triang2d` instance `bnd_curve->geom` to `bnd_curve->x/y/z`. This will update the position of the points if the `geom` surface the `bnd_curve` belongs to was changed.

GRAPE(bnd_curve, "interpol")(bnd1, bnd2, factor)
GRAPE(bnd_curve, "interpol-fac")(bnd1, bnd2, fac1, fac2)

BND_CURVE *bnd_curve, *bnd1, *bnd2
double factor, fac1, fac2

Same as `"interpol"`/`"interpol-fac"` on `Triang1d` (see 6.2.1.4). Additionally the `geom` entry of `bnd_curve` is set to `NULL` because the associated `Triang2d` instance for `bnd_curve` is not known. The `vertex` and `neighbour` lists are copied from `bnd1`.

GRAPE(bnd_curve, "make-neighbour-send")()

BND_CURVE *bnd_curve

Compute the `neighbour` values, i.e. the indices of the triangles of the `geom` surface along the curve.

GRAPE(bnd_curve, "merge")(bnd_two)

BND_CURVE *bnd_curve, *bnd_two

Merge two `Bnd_Curve` instances: append the lists of `bnd_one` to those of `bnd_two`. If the method fails it returns `NULL`.

GRAPE(bnd_curve, "reflect")(trfmat, orient)

BND_CURVE *bnd_curve
MATRIX44 trfmat
int orient

Create a reflected `Bnd_Curve` instance. Same as `Triang1d` `"reflect"` but additionally the `vertex` and `neighbour` lists are copied. The reflected `Bnd_Curve` instance is returned.

6.2.3 Cons

Instances of this class always belong to a geometry because they don't have their own coordinate arrays. For a description of the methods which are not documented here see `Triang1d`. The methods are almost identical to the `Triang1d` methods, the only real difference is that instead of `triang->x/y/z[i]` one has to use `cons->geom->x/y/z[cons->vertex[i]]`.

```
#CLASS Cons : Root {
  TRIANG2D *geom;            /* pointer to geometry to which cons belongs,
                                it should only be changed with "set-geom" */
  int number_of_points;      /* number of points */
  int max_number_of_points;  /* maximum number of points */
  int *vertex;               /* indices of the points in geom */
  int flag;                  /* flag for ACTIVE, DISPLAY etc */
};
```

6.2.3.1 Memory Methods on Cons

GRAPE(cons, "set-geom")(triang)

CONS *cons
TRIANG2D *triang

This method should be called directly after `"new-instance"` to assign a geometry to the curve `cons`, without this most other methods won't work. This method handles the `refcount` of the curve's geometry, i.e. an old `cons->geom` is deleted first, the new geometry `triang` then is `ASSIGN`ed.

GRAPE(cons, "free")()

CONS *cons

GRAPE(cons, "list-alloc")(max_points)
GRAPE(cons, "list-free")()

CONS *cons
int max_points

GRAPE(cons, "list-shrink-send")()
GRAPE(cons, "optimize-mem-send")()

CONS *cons

GRAPE(cons, "hardcopy")()
GRAPE(cons, "softcopy")(copy)

CONS *cons, *copy

6.2.3.2 Display Methods on `Cons`

GRAPE(cons, "display")()
GRAPE(cons, "tube")(discr, scale)

> **CONS *cons**
> **int discr**
> **double scale**

Like `"display"` on `Bnd_Curve` this method only exists to provide a new ruler set for `Cons` instances.

6.2.3.3 I/O Methods on `Cons`

GRAPE(cons, "xdr")(xdrp)

> **CONS *cons**
> **XDR *xdrp**

6.2.3.4 Debug Methods on `Cons`

GRAPE(cons, "get-info")(xlist)

> **CONS *cons**
> **XLIST *xlist**

GRAPE(cons, "show-send")()

> **CONS *cons**

6.2.3.5 Other Methods on `Cons`

GRAPE(cons, "merge")(cons_two)

> **CONS *cons, *cons_two**

Merge `cons_two` into `cons` by appending the `vertex` indices. Of course this is only possible if both instances point to the same `geom` and if the last index of `cons` and the first index of `cons_two` are identical.

GRAPE(cons, "close-send")()
GRAPE(cons, "verify-use")()

> **CONS *cons**

After merging and closing two `Triang2d` (most likely `Geom2d`) instances some of the points the `cons` uses may no longer be used by the `cons->geom`, they will no longer be updated and could be removed. `"verify-use"` checks whether all points are still in use, `"close-send"` will replace unused points by the points they were identified with during a `"close"` on `cons->geom`.

(TRIANG1D *) GRAPE(cons, "get-triang")(triang)

> **CONS *cons**
> **TRIANG1D *triang**

Copy the coordinates of the `cons` from `cons->geom->x/y/z` with indices `cons->vertex` to `triang ->x/y/z`. The `triang` instance is returned, if `triang` is `NULL`, a new `Triang1d` instance is created.

GRAPE(cons, "interpol")(cons1, cons2, factor)
GRAPE(cons, "interpol-fac")(cons1, cons2, fac1, fac2)

> **CONS *cons, *cons1, *cons2**
> **double factor, fac1, fac2**

GRAPE(cons, "reflect")(trfmat, orient)

> **CONS *cons**
> **MATRIX44 trfmat**
> **int orient**

GRAPE(cons, "reset")()

> **CONS *cons**

6.3 Surfaces and 2D Finite Elements

The classes in this chapter are needed to work with any type of triangulated simplicial complexes. The simplest class Triang2d only contains information about the coordinates, the connectivity and neighbourhood relationship between triangles in case of regular triangulations.

The subclass Fe2d contains additionally information necessary for working with finite element solutions of partial differential equations. The function `f` returns in its parameter vector `value` a vector of dimension `dim` at every point on the surface. As an example we may choose `dim = 1` and `f` returning the z-component of each point, i.e. `f` gives the height of each point. Now using the display method `"isoline-disp"` we obtain height level lines on the surface.

The subclass Adapt2d is designed to control adaptive refining and coarsening of triangulations and the handling of boundary functions. To use the refining and coarsening on user defined data the functions `interpol`, `restrict`, `element_compress` and `point_compress` can be defined.

A further subclass is `Geom2d`, it contains additional instance variables for the normal vector at each vertex, for RGB colors at each vertex or triangle and for boundary curves and parameter lines. All instance variables are optional, they are only allocated on request or if they are needed by a method (`"display"` in patch mode needs for examples normals). `Geom2d` instances are used by class `Surface` (see 7.4.3) and its subclasses to build up complex time-dependent geometries.

6.3.1 Triang2d

```
#CLASS Triang2d : Root {
  int number_of_points;
  int max_number_of_points;
  double *x, *y, *z;          /* point coordinates, z==NIL possible */
  int number_of_elements;
  int max_number_of_elements;
  INT3 *vertex;               /* vertex indices for all triangles */
  INT3 *neighbour;            /* indices of neighbour triangles */
                              /* neighbour[j][i] opposite to vertex[j][i] */
  int data_is_borrowed;
};
```

```
typedef INT3 int[3];
```

Like all triang classes `Triang2d` has a three coordinate vectors `x`, `y` and `z` which define the location of points in three-space, the coordinates may be restricted to two dimensions by setting the third coordinate pointer `z` to `NULL` (methods like `"display"` which need a third coordinate will use zero as default). But be careful when using this feature with `Geom2d` and its subclasses, some methods may not be able to handle this `NULL` pointer.

The variable `vertex` points to an array which contains the global vertex numbers for the local vertices of the triangles, the `neighbour` variable contains information of the neighbour elements. Example:

```
              14 +-------+ 13
                 |\2    1|
                 |1\     |
                 |  \    |
triangle 0 -->   |   \   | <-- triangle 1
                 |    \  |
                 |     \0|
                 |2    0\|
              11 +-------+ 12
```

The local vertices of each triangle are numbered 0, 1 and 2, the numbers 11, 12, 13 and 14 are the global vertex numbers corresponding to the local ones. For this configuration the variables have the following values:

```
vertex[0][0] = 12           vertex[1][0] = 12
vertex[0][1] = 14           vertex[1][1] = 13
vertex[0][2] = 11           vertex[1][2] = 14

neighbour[0][0] = -1        neighbour[1][0] = -1
neighbour[0][1] = -1        neighbour[1][1] = 0
neighbour[0][2] = 1         neighbour[1][2] = -1
```

The `neighbour` array contains for each triangle for each local vertex the number of the neighbour of the edge opposite to the vertex, negative values mark boundary edges.

6.3.1.1 Memory Methods on `Triang2d`

GRAPE(triang2d, "free")()

TRIANG2D *triang2d;

The associated memory of the instance variables is freed and the instance is deleted. For a more detailed description of this method see 5.4.4.1. This method is normally only called by the delete-method on class ROOT.

GRAPE(triang2d, "list-alloc")(max_points, max_elements)

GRAPE(triang2d, "list-growto")(max_points, max_elements)

GRAPE(triang2d, "list-free")()

TRIANG2D *triang2d;
int max_points, max_elements

`"list-alloc"` allocates memory for `max-points` points for the vertex lists and for `max_elements` elements for the connectivity, `"list-free"` frees the memory for the vertex lists and for the connectivity.

`"list-growto"` uses `"list-alloc"` to assure at least `max_...` points and elements exist.

For further information about list-alloc see 5.4.4.2.

GRAPE(triang2d, "point-list-alloc")(max-points, tri-connect)

GRAPE(triang2d, "point-list-free")()

TRIANG2D *triang2d;
TRIANG2D *tri-connect;
int max-points;

"point-list-alloc" allocates memory for max-points points for the vertex lists. If tri-connect is not NULL, it allocates memory for the connectivity too. "point-list-free" frees only the memory for the vertex lists.

GRAPE(triang2d, "list-shrink-send")()

TRIANG2D *triang2d;

Free unused space of the vertex lists: new vertex lists for triang2d->number_of_points points are allocated, the points with index from 0 to triang2d->number_of_points are copied and the old vertex lists are freed. This method is called by

GRAPE(triang2d, "optimize-mem-send")()

TRIANG2D *triang2d

Free unused space of the vertex lists.

GRAPE(triang2d, "hardcopy")()

TRIANG2D *triang2d;

Create an identical copy of triang2d which has its own lists (all data is copied). "hardcopy" may not be sent to superclasses since always a new structure is allocated. The pointer to the new Triang2d instance is returned.

GRAPE(triang2d, "softcopy")(t2-copy)

TRIANG2D *triang2d;
TRIANG2D *t2-copy;

Make a softcopy of a Triang2d instance: like "hardcopy" but only the used data (up to number-of-points and number-of-elements) is copied. If t2-copy is not NULL then this instance is used to store the copy otherwise a new Triang2d instance is created.

(int *)GRAPE(t2-one, "merge")(t2-two)

TRIANG2D *t2-one;
TRIANG2D *t2-two;

Merge two Triang2d instances: append the vertex lists of t2-one to those of t2-two. If the method fails it returns NULL.

6.3.1.2 Display Methods on `Triang2d`

GRAPE(triang2d, "display")()

TRIANG2D *triang2d;

The Triang2d instance is rendered on the standard device in a default manner.

GRAPE(triang2d, "clip-ball-disp")()

GRAPE(triang2d, "clip-plane-disp")()

(TRIANG2D *)GRAPE(triang2d, "fix-clip-send")()

GRAPE(triang2d, "fix-clip")(f, var)

GRAPE(triang2d, "clip")(f, var)

TRIANG2D *triang2d ; /* set of triangles to be clipped */
double (*f)() ; /* pointer to clipping function */
void *var ; /* pointer to parameters of clipping function */

The methods are designed to support general clipping facilities with varying hypersurface and local rendering function. A set of triangles is divided by the implicitly given clipping function f into a visible (f > 0) and an invisible (f < 0) part. The zero level of this clipping function defines the clipping level.

The function f must be of the following form:

```
f(double x, double y, double z, char *var)
```

The variable var in the parameter list of the method call and of the function f must be of the same type. *var determines the clipping function. The implicit function description of a plane or a ball are predefined for interactive use:

```
double plane(double x, double y, double z, PLANE_PARM *var)

typedef struct {
  double n_x, n_y, n_z;
  double distance;
} PLANE_PARM; /* Hesse Form */

double ball(double x, double y, double z, BALL_PARM *var)

typedef struct {
  double center_x, center_y, center_z;
  double radius;
} BALL_PARM;
```

The "fix" methods fix the clipping and give it back in the Triang2d instance.

GRAPE(triang2d, "shrink-disp")()

GRAPE(triang2d, "shrink-foc-tria-disp")()

GRAPE(triang2d, "shrink-foc-point-disp")()

GRAPE(triang2d, "shrink-layer-disp")()

GRAPE(triang2d, "shrink-angle-disp")()

GRAPE(triang2d, "shrink")(p, f, fpar)

TRIANG2D *triang2d;
double p;
int (*f)();
char *fpar;

The set of tetrahedron belonging to the Triang2d instance is displayed on the standard device as a patch model. It is possible to identify each triangle because they are separated from each other through a small space given by the parameter p. There is a selection criterion for those elements which are of interest. The specific triangle is displayed if f(tr, i, fpar) is YES. There are several interactive versions with different functions f:

"shrink-layer-disp"

displays a layer of triangles whose centers lie in a layer surrounding a certain plane.

"shrink-angle-disp"

displays those elements with the spatial angles between two bounds.

"shrink-foc-point-disp"

displays the neighbourhood of a certain point with the index given by a ruler variable.

"shrink-foc-tria-disp"

does the same for an element index.

GRAPE(triang2d, "points-disp")()

GRAPE(triang2d, "points")(radius)

TRIANG2D *triang2d;
double radius;

The Triang2d instance is rendered as a cloud of points. The ball radius is controlled by a ruler which is added to the manager by the interactive method "points-disp".

6.3.1.3 I/O Methods on `Triang2d`

(TRIANG2D *) GRAPE(Triang2d, "read")()

CLASS *Triang2d

Read a `Triang2d` instance from the standard textdevice. If successful the instance otherwise `NULL` is returned. This method is only provided for backward compatibility to read files written with `"write"`, it is replaced by

GRAPE(triang, "xdr")(xdrp)

TRIANG2D *triang
XDR *xdrp

Read the instance `triang` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

6.3.1.4 Other Methods on `Triang2d`

GRAPE(tr2d, "check-send")()

TRIANG2D *tr2d ;

Checks consistency of a triangulation, calls "check-phase" with $phase = 6$.

GRAPE(tr2d, "check-phase")(phase)

TRIANG2D *tr2d;
int phase;

This method invokes the checking of specific data in the geometry. There are the following possibilities at hand (a bigger number includes all the lower ones):

- Phase 0 - global variables
- Phase 1 - vertex field
- Phase 2 - neighbour field
- Phase 3 - connectivity
- Phase 4 - Euler-Poincare-Characteristic
  - a - full volume (triangulation)
  - b - boundary (edges)
- Phase 5 - location of points with respect to common face of neighb. triangle

- Phase 6 - volume

GRAPE(triang, "interpol")(tr1, tr2, factor)
GRAPE(triang, "interpol-fac")(tr1, tr2, fac1, fac2)

> **TRIANG2D *triang, *tr1, *tr2**
> **double factor, fac1, fac2**

Linear interpolation of the two `Triang2d` instances `tr1` and `tr2` with `factor`:

```
triang = (1 - factor) * tr1 + factor * tr2
```

or with factors `fac1` and `fac2`:

```
triang = fac1 * tr1 + fac2 * tr2.
```

If necessary new vertex lists are allocated for `triang`.

The two `Triang2d` instances are interpolated pointwise, therefore they must have the same `number_of_points` and the same connectivity `vertex` and `neighbour`.

GRAPE(triang, "center-send")()
GRAPE(triang, "normalize-send")()

> **TRIANG2D *triang**

`"center-send"` moves `triang` to make the minimum and maximum x/y/z-coordinates identical (apart from the sign), i.e. it centers `triang` at zero. `"normalize-send"` additionally scales `triang` so that it fits into the [-1, +1] cube.

GRAPE(triang2d, "reflect")(trfmat, orient)

> **TRIANG2D *triang2d;**
> **MATRIX44 trfmat;**
> **int orient;**

Create a new Triang2d instance from triang2d with triang2d->number_of_points points which are transformed with trfmat. If trfmat is NULL the points are copied without transformation. The orientation flag orient is not needed but supplied so that all reflection methods have identical parameter lists. The reflected Triang2d instance is returned.

GRAPE(triang2d, "show-send")()

> **TRIANG2D *triang2d;**

Print the data of a Triang2d instance to stdout.

GRAPE(triang2d, "c-show")()

> **TRIANG2D *triang2d;**

Write the data of the triang2d instance in C source format to the standard output.

GRAPE(triang2d, "get-info")(list)

> **TRIANG2D *triang2d;**
> **LIST *list;**

Create info list entries for a Triang2d instance. Entries are class, name, refcount, number of points, maximum number of points, number of triangles and maximum number of triangles.

GRAPE(triang2d, "reset")()

> **TRIANG2D *triang2d;**

Reset triang2d->number_of_points and triang2d->number_of_elements to zero (only needed in ->universal mode).

GRAPE(triang2d, "center")()

TRIANG2D *center;

The centerpoint of all points of the whole triangulation is set to 0.0. This helps to manage the object in the interactive environment.

GRAPE(triang2d, "refine4-send")()

TRIANG2D *triang2d;

The triangulation is refined once by the standard subdivision into 4 triangles.

GRAPE(triang, "find-pick")(v, u, w, obj, dist)

GRAPE(triang, "find-pick-index")(v, index, u, w, obj, dist)

TRIANG2D *triang;
VEC3 v;
int *index;
double u, w, *dist;
TRIANG2D **obj;

Search for a point v in triang for which the distance from the straight line (u, w, .) to PVM * v (the point v transformed by the PVM matrix) is less than sqrt(*dist). If we find such a point, *obj is set to triang, v to the new point transformed by the MODEL matrix and *dist to the new distance. "find-pick-index" additionally sets *index to the index of the point.

For an explanation of the transformation matrices see 9.2.2.1.

GRAPE(triang, "find-pick-intersec")(v, u, w, element, bary)

TRIANG2D *triang;
VEC3 v ; /* point, which is searched */
double u, w ; /* obtained values from the */
/* method "get-pick-pos" */
int *element ; /* number of triangles, which */
/* contains the determined point */
VEC3 bary ; /* barycentric coordinates of the determined point */

This methods finds on the current plane from the current clipping method the intersection point of the ray (u, w, .) in screen coordinates where u and w are between –1 and 1. Afterwards the coordinates of the vector, the picked element and the barycentric coordinates are v, element and bary. The method returns the triangulation if an intersection could be found otherwise NULL.

GRAPE(Triang2d, "mask-make")(x, y, x_n, y_n, mask, inst-name)

CLASS *Triang2d;
double *x, *y;
int x_n, y_n;
int *mask;
char *inst-name;

This method generates a Triang2d instance containing the triangulation of a planar rectangular domain. This domain consists of those cells which are marked by a YES in the compressed 2D $(x\_n - 1) \times (y\_n - 1)$ array Mask[x][y] == mask[(x - 1)*y_n + y - 1]. If the user wishes to define non standard coordinates he gives them in form of the 1D arrays x[x_n] and y[y_n]. Otherwise (x == NULL) a equal distant subdivision of the line [0, 1] is used.

GRAPE(Triang2d, "explicit")(name, x, y, z, ax, ex, ay, ey, dx, dy)

char *name ; /* name of instance */
double *x, *y, *z; /* pointer to functions of coordinates */
double ax, ex; /* minimum and maximum of the first parameter */
double ay, ey; /* minimum and maximum of the second parameter */
int dx, dy; /* number of discretization in the parameter domains */

A Triang2d instance is built up for the parameterized surface, which is given through the coordinate functions x, y and z. Compare project *Explicit* 8.2.5, which allows to compute time–dependent surfaces reading the functions from a data file.

GRAPE(triang, "mollify-send")()

TRIANG2D *triang2d;

A triangulation is mollified, so that the distances of triangulation points are more regular and the surfaces are smoother.

(TRIANG3D *)GRAPE(triang, "simple-rot")(name, n, fi)

(TRIANG3D *)GRAPE(triang, "rot")(name, n, fi, t2t, p2p)

TRIANG2D *triang;
char *name ; /* name of produced Triang3d instance */
int n ; /* number of angles */
double fi ; /* angle in radiant */
int **t2t ; /* relation tetrahedron to triangle */
int **p2p ; /* relation 3d-point to 2d-point */

A set of tetrahedron, which are axissymmetric disposed, is constructed from a planar 2D triangulation: rotating each triangle about an angle with fi in radiant in the positive direction around the z–axis starting with the x/y–plane yields a prism–like object, division of this prism into four tetrahedron. This construction can be repeated n times. In case of a closed circle (n * fi = 2 * pi) the connectivity at the ends is generated. For "rot" the user has to provide two pointer to integer arrays which on return contain for each tetrahedron/point the index of the triangle/point it was build from.

Note that the triangles must be positive oriented in order to use these methods.

(TRIANG2D *)GRAPE(tr,"level")(funct,lev)

(TRIANG2D *)GRAPE(tr,"fix-super-level")(funct,lev)

(TRIANG2D *)GRAPE(tr,"fix-sub-level")(funct,lev)

TRIANG2D *tr;
void (*funct)();
double lev ;

'level' calculates the isoline of the given finite element function funct on the level lev. 'fix-super-level' creates the superniveau fixed by the actual value of level as a new triang2d instance. 'fix-sub-level' creates the subniveau.

GRAPE(triang2d, "edges")(angle)

GRAPE(triang2d, "show-edges-send")()

TRIANG2D *triang2d;
double angle;

If the triangulation is a surface in 3d it can be a problem since the triangle-normals are interpolated also at the edges, which therefore appear blurred. In using these methods additional fictive points are inserted at the edges, which prevents the interpolation. The minimal edge-angle only just specifying an edge is the parameter angle, which can also be set up with the root-method "choice-edge-angle-send".

(SCENE *)GRAPE(s,"split-surfaces-send")()

SCENE *s ; /*of which the object is 2D*/

With this method as well the interpolation of the normals at the edges is avoided by dividing the triangulation at the edges. For each singular surface a new scene appears in the scene tree. The minimal edge-angle can be set up using the root-method "choice-edge-angle-send".

(VEC3 *)GRAPE(triang2d, "get-vertex-wnormals")(vnorm)

TRIANG2D *triang2d;
VEC3 *vnorm;

Compute weighted vertex normals for a Triang2d instance. If vnorm == NULL the needed memory is allocated. Normals are weighted with the angle of the triangle the normal belongs to. "get-vertex-wnormals-send" of Geom2d uses this method.

(VEC3 *)GRAPE(triang2d, "get-triangle-normals")(tnorm)

TRIANG2D *triang2d;
VEC3 *tnorm;

Compute triangle normals for a Triang2d instance. If vnorm == NULL the needed memory is allocated.

(VEC3 *)GRAPE(triang2d, "draw-normals-disp")()

TRIANG2D *triang2d;

Draws triangle normals for every triangle of a Triang2d instance. The length of the normals may be scaled by a ruler.

(TRIANG2D *)GRAPE(tr2,"orient-send")()

(TRIANG2D *)GRAPE(tr2,"inverse-orient-send")()

(TRIANG2D *)GRAPE(tr2,"switch-orient-send")()

TRIANG2D *tr2;

These methods concern the local orientation of the triangles. orient-send checks if the orientation of the triangles is always in the same direction. If not, the orientation of the triangle no.0 is taken to all triangles. inverse-orient-send changes the orientation of the whole triangulation (e.g. inner normals get outer normals). This is done by changing the orientation of triangle no.0 and calling orient-send.

GRAPE(triang2d, "sort-points-send")()

TRIANG2D *triang2d;

"sort-points-send" sorts points such that all used points are at the beginning of the point list.

GRAPE(triang2d, "regenerate-send")()

TRIANG2D *triang2d;

"regenerate-send" regenerates a degenerate triang2d structure by removing triangles with very short edges, double triangles or triangles with very short angles in this order, but if necessary in a multiple loop. After this method has been called, there may exist a lot of unused points in the triang2d->x/y/z point list.

GRAPE(tr2d, "identify-nbs-send")()

GRAPE(tr2d, make-neighbour-send")()

TRIANG2D *tr2d;

These methods build neighbourhood information for an arbitrary `tr2d` only out of `tr2d->vertex` information, using different algorithms. If not existent the memory for `tr2d->neighbour` is allocated.

(LIST_OF_INST *)GRAPE(triang2d, "make-edge-conn")()

TRIANG2D *triang2d;

"make-edge-conn" creates connectivity information for edges: The return list (LIST_OF_INST *)loi will contain for each vertex a List_Of_Int instance with the indices of all its adjacent points. The length of each vector equals the number of adjacent points:

```
loi->num_of_objects = triang->number_of_points;
loi->objects[i] = (LIST-OF-INT *)vector;
```

with

```
vector->num_of_int = number of adjacent points;
vector->vec = (int *)vec[number of adjacent points];
```

(LIST-OF-INST *)GRAPE(triang2d, "make-triangle-conn")()

TRIANG2D *triang2d;

"make-triangle-conn" creates connectivity information for triangles: The return list LIST-OF-INST *loi will contain for each vertex a List-Of-Int instance with the indices of all its adjacent triangles. The length of each vector equals the number of adjacent triangles:

```
loi->num_of_objects = triang->number_of_points;
loi->objects[i] = (LIST_OF_INT *)vector;
```

with

```
vector->num_of_int = number of adjacent triangles;
vecto->vec = (int *)vec[number of adjacent triangles];
```

GRAPE(triang, "equiangulate-send")()
GRAPE(triang, "equiangulate-s3-send")()

TRIANG2D *triang

Equiangulate `triang` by switching diagonals between ajacent triangles if the angles opposite to the diagonal add up to more than `pi`. This criterion if equivalent to `cos(a) + cos(b) < 0`, the cosines are calculated by the law of cosines. This is done in a loop over all edges, each switch of a diagonal reduces the radius of the circumcircles of the two triangles, therefore the loop will terminate. To prevent infinite loops `-eps` is used insted of `0` for comparison. The s3-method does the same but the points are projected to the three-sphere first.

This algorithms can be applied both to planar triangulations and triangulations in three-space. In the second case some switches might have to be skipped to avoid degenerations of the triangulation. The number of loops, switched diagonals and skipped switches are printed to `stdout`.

6.3.2 Fe2d

```
#CLASS Fe2d : Triang2d {
  int dimension_of_value; /* 1==scalar,  >1 => vector function     */
  int polynomial_order;   /* 1==linear, 2==quadratic,... elements  */
  void (*f)(FE2D*, int, VEC3, double*);
                          /* void *f( FE2D *fe, int triangle,
                                      VEC3 barycentric_coordinates,
                                      double value[dim] )          */
  /* optional ****************************************************** */
  int size_of_data;                         /* size of user data     */
  char *data;                               /* and pointer to it     */
  /* ************************************************************** */
};
```

6.3.2.1 Memory Methods on Fe2d

GRAPE(fe2d, "free")()

FE2D *fe2d;

Free the memory lists and the userdata of the Fe2d instance and delete the instance itself. For further details see 5.4.4.1. This method is normally only called by the delete-method on class ROOT.

GRAPE(fe2d, "hardcopy")()

FE2D *fe2d

Implementing "hardcopy".

GRAPE(fe2d, "softcopy")(copy)

FE2D *fe2d, *copy

Implementing "softcopy".

6.3.2.2 Display Methods on Fe2d

GRAPE(fe, "graph-disp")()

FE2D *fe

Display the function value as a graph.

GRAPE(fe, "isoline-disp")()

GRAPE(fe, "isoline")(select, funct, linescal, colscal, offsetscal, highscal, max, min)

FE2D *fe;
int select; /* = 1 for "isoline-select", else 0 */
void (*funct)(); /* function specifying the level lines */
double linescal; /* density of lines */
double colscal; /* line color */
double offsetscal; /* offset */
double highscal, max, min;

Level lines of the function funct supported by the 2D triangulation of the Fe2d instance are calculated. The function can be arbitrary carrying the argument structure of Finite Element functions (see listing of struct fe2d above), if the pointer is set to NULL the absolute value of the Fe2d function is used. Depending on whether the device is in the patch or in the grid mode colored patches respectively lines are displayed. The color is generated according to a ramp in the HSV color model.

The parameters for the color model are ln(range of colormap) and ln(position of colormap). The other parameters are ln(line-distance), the distance of the isolines (only used in grid mode), and the height (i.e. z–value) of the isolines. For this method it is necessary to set the dimension_of_value and the polynomial_order entries in the structure.

GRAPE(fe, "isoline-select-disp")()

FE2D *fe;

Those particular values which are stored in the array colorbar->values (see section 6.6) are displayed. The colorbar is the one belonging to the function "fe2d-isoline-disp". The values can be chosen with a colorbar-method or with the prompt "add value" (which appears after clicking the button "iso2dclb modi"). In grid mode only these particular isolines are displayed, whereas in patch mode the values are shown by means of white lines above the colored patch image.

GRAPE(fe, "line-graph-disp")()

FE2D *fe

Display the function value as a line-graph.

GRAPE(s, "mark-polygon-send")()

SCENE *s ;

With this method a polygon in the triangulation can be determined in the following way:

- locate some points in the triangulation;
- pick them by pressing the left mouse button;
- mark the last polygon point with pressing the middle mouse button.

Picking means in the case of a Triang2d instance as the object of the scene s the direct picking of points on the triangulation. If s->object is an instance of a superclass of Triang3d picking is performed on the current planar clipping plane (see methods clip-....-disp). The picked polygon is saved as a Triang1d instance when pressing the right mouse button. This Triang1d instance is inserted as the object of a second scene node stored as s->next_scene. This means it is inserted as the following hierarchical node.

GRAPE(fe, "vect-disp")()

GRAPE(fe, "vect")(length, mesh, dist, color_scal, offset_scal, max, min)

FE2D *fe;
double length , mesh, dist;
double color_scal, offset_scal, max, min;

In the case of fe->dimension_of_value ≥ 2 the values of the Fe2d function are shown at the points of a constructed rectangular mesh with mesh width mesh. The vectors' length can be regulated with the parameter length. dist is the distance of mesh from the object.

- If fe->z $\neq$ NIL the direction of the vectors is three-dimensional. In that case fe-> dimension_of_value > 2 is important.
- If fe->z = 2 and fe-> dimension_of_value = 2 the vectors are all of one color and are displayed within one plane. The color is either white or grey depending on the lightness of the object.
- If fe->z = 2 and fe-> dimension_of_value ≥ 2 the color of the vectors depends accordingly on the values that are calculated in the third component of the function fe->f. The colors are generated with the hsv color model and can be changed with the parameters color, offset, max and min (of the corresponding colorbar vect2dclb).

6.3.2.3 I/O Methods on Fe2d

(FE2D ∗) GRAPE(Fe2d, "read")()

CLASS ∗Fe2d

This methods extends the superclass method, it additionally reads the finite element data, the corresponding finite element function is treated as linear. If successful the instance otherwise `NULL` is returned. This method is only provided for backward compatibility to read files written with `"write"`, it is replaced by

GRAPE(fe, "xdr")(xdrp)

FE2D ∗fe
XDR ∗xdrp

Read the instance `fe` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

The `fe->data` can only be handled if its size is a multiple of `sizeof(double)`, i.e. when only `double` data for each vertex or element is stored in this pointer. The finite element function `f` is not handled by this function, it has to be set by the user after the data has been read in. See section 5.7.7 for more information.

6.3.2.4 Other Methods on Fe2d

GRAPE(fe2d, "point-probe-send")()

FE2D *fe2d;

The value of the FE function at a position the user has to pick with the mouse is displayed.

(FE3D *)GRAPE(fe2d, "rot")(name, n, fi, t2t, p2p)

FE2D *fe2d;
char *name ; /* name of produced Triang3d instance */
int n ; /* number of angles */
double fi ; /* angle in radiant */
int **t2t ; /* relation tetrahedron to triangle */
int **p2p ; /* relation 3d-point to 2d-point */

The parameters have the same meaning as in the method "rot" on a Triang2d instance (6.3.1.4).

Additional to the building of the tetrahedron from the corresponding Triang2d instance a 3D finite element function is constructed with the Fe2d instance function.

GRAPE(fe2d, "set-dim-of-value")(value)

FE2D *fe2d;
int value;

"set-dim-of-value" sets fe2d->dimension_of_value = value.

GRAPE(fe2d, "interpol")(fe1, fe2, factor)

FE2D *fe2d;
FE2D *fe1, *fe2;
double factor;

Same as "interpol" on Triang2d (6.3.1.4). Additionally the Finite Element function data is interpolated. This only takes place when the function fe2d->f is used, therefore this method is as fast as "interpol" on the superclass Triang2d, but for evaluating the FE function on the interpolated instance fe2d (e.g. in "isoline") both fe1->f and fe2->f have to be evaluated.

GRAPE(fe2d, "get-info")(xlist)

FE2D *fe2d;
XLIST *xlist;

Create info xlist entries for a Fe2d instance. Entries are class, name, number of points, maximum number of points, number of triangles, maximum number of triangles, dimension of value and polynomial order.

GRAPE(fe2d, "get-dim-of-value")(dimp)

FE2D *fe2d
int *dimp

Fill `*dimp` with the dimension of value.

GRAPE(fe2d, "get-min-max")(minimum, maximum)

FE2D *fe2d
double *minimum, *maximum

Get the minimal and maximal function value. This method is implemented on any class containing a function. It can be used adjusting the `Colorbar`.

6.3.3 `Adapt2d`

```
#CLASS Adapt2d : Fe2d {
  REFINE2D *refine;
  int number_of_bnd_functions; /* (max) no of different boundaries */
  int max_number_of_bnd_functions;
  BND_FUNCTION *bnd_function;   /* bnd_function[-neighbour[i][j]]  */
                                /*  ->g(x,bnd_function->par) = 0.0 */
                                /* is the boundary   (VEC3 x)      */
    /* Functions for interpolation and restriction
     * calls: interpol(ADAPT2D *tr,
     *                 int oldtriangle, int oldpoint1, int oldpoint2,
     *                 int newpoint, int newtriang1, int newtriang2)
     * oldpoint1 and oldpoint2 are the global points of the divided edge
     * newpoint is the global number of the new (added) point)
     *
     *        restrict(ADAPT2D *tr,
     *                 int deletriang1, int deletriang2, int deletpoint,
     *                 int newtriang, int makropoint1, int makropoint2)
     * deletpoint is the global number of the deleted point
     * makropoint1 and makropoint2 are the global points of the
     * coarsened edge in the new triangle
     *
     * while coarsening, gaps in the triangulation-arrays have to be
     * filled by moving elements or points with:
     *         element_compress(ADAPT2D *tr, int old_element_number,
     *                          int new_element_number)
     *         point_compress(ADAPT2D *tr, int old_point_number,
     *                          int new_point_number)
     */
  void (*interpol)(), (*restrict)();
  void (*element_compress)(), (*point_compress)();

};
```

```
typedef struct refine2d {
  unsigned char refine;       /* flag to refine/coarse */
  unsigned char edge;         /* next refinement edge */
  unsigned char friend;       /* local number of the last neighbour */
  unsigned char level;        /* level of refinement */
  unsigned char vtx_level[3]; /* generation of vertices */
} REFINE2D;


typedef struct bnd_function {
  double (*g)();
  void *par;
    /* bnd_function[-neighbour[i][j]->g(VEC3 x, bnd_function->par) = 0.0
       is the boundary with x */
} BND_FUNCTION;
```

6.3.3.1 Memory Methods on `Adapt2d`

(ADAPT2D *)GRAPE(Adapt2d, "new-instance")(name)

CLASS *Adapt2d;
char *name;

Creates an instance of class Adapt2d with name name. Only the struct itself is allocated. All other memory has to be allocated separately.

6.3.3.2 Display Methods on `Adapt2d`

GRAPE(adapt, "refine-display")()

ADAPT2D *adapt

This method shows which edges would be refined by calling `"refine"`. These edges are drawn in red, the new edges that would be inserted are drawn in yellow. The result of a `"refine"` can differ slightly from what this display method shows, because correction steps of the `"refine"` method are not taken into account.

GRAPE(ad2d, "deg-fine-disp")()

ADAPT2D *ad2d;

A display method to show the refining level `refine[i].level` for each triangle in the triangulation. The color of the shrinked displayed frame/patch (according to grid/patch) for each triangle is red for level 0 and yellow for the maximal refine–level which can be adjusted with a ruler.

6.3.3.3 I/O Methods on `Adapt2d`

GRAPE(adapt, "xdr")(xdrp)

ADAPT2D *adapt
XDR *xdrp

Read the instance `adapt` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

Only the `refine` field is handled by this function, like for `Fe2d` the function pointers has to be set by the user after the data has been read in.

6.3.3.4 Other Methods on `Adapt2d`

GRAPE(ad2d, "refine-global-send")()

GRAPE(ad2d, "refine-local")()

GRAPE(ad2d, "coarsen-global-send")()

GRAPE(ad2d, "coarsen-local")()

GRAPE(ad2d, "set-rfc-field")(element)

GRAPE(ad2d, "set-rfc-field-send")()
    ADAPT2D *ad2d;
    int element;

We use a refinement algorithm due to Bänsch, who has developed a corresponding algorithm in 3D, too. A similar approach by Rivara is restricted to the 2D case.

In this algorithm the refining / coarsening a triangulation means to reverse previously executed refinements. Therefore it is necessary to store additional information for each triangle. To accomplish this we enlarge the FE2D data structure by the structure REFINE2D (see 6.3.3). This structure includes the flag to refine or to coarse with possible values `0, G_REFINE` or `G_COARSE`, the refine edge and the level of refining. In the macro triangulation, where no triangle is refined, edge is the local number of the edge to refine (in general the longest edge) in each triangle and level is zero. In the situations that can be coarsened all triangles must have the same refinement level. Using this additional information in the REFINE2D structure a unique coarsening is possible.

In this context it is important to mention, that only the refine flag should be set by the user. Before refining the triangulation the first time, it is also possible to set the refine edge. In general the initialization is done by using "set-rfc-field-send" which allocates the memory for the REFINE2D structure and sets the refine edge on the longest one.

"refine-global-send" bisects all triangles (including the in general necessary correction) in the triangulation according to their refinement edge. If no refine structure is set, "set-rfc-field-send" is executed at first.

"refine-local" bisects all with `G_REFINE` in the refine flag marked triangles according to their refinement edge and makes the necessary corrections. The refine flag in the REFINE2D structure is not initialized after the local refining and can be used again or must be reset to 0 before starting "refine-local" again.

"coarsen-global-send" coarsens all refined triangles if possible. In most cases "coarsen-global-send" must be used several times to reverse all previously executed refinements.

"coarsen-local" coarsens all with `G_COARSE` in the refine flag marked triangles if they are the result of a previously executed refinement. The refine flag in the REFINE2D structure is not initialized after the local coarsening and can be used again or must be reset to 0 before starting "coarsen-local" again.

If boundary segments are refined which belong to a boundary curve stored in the structure BND_FUNCTION, "fit-bndfunctions-send" (see 6.3.3.4) is executed automatically.

To handle the data on the triangulation there are four functions, which may be defined by setting the corresponding function pointer in the ADAPT2D structure. They are executed during the refining and coarsening:

interpol (ADAPT2D *ad2d, int oldtriangle, int oldpoint1, int oldpoint2, int newpoint, int newtriang1, int newtriang2), where oldpoint1 and oldpoint2 are the global points of the divided edge and newpoint is the global number of the new (added) point. In this function the user can interpolate his own data and do the memory management for this. It is executed in every refinement.

restrict (ADAPT2D *ad2d, int deletriang1, int deletriang2, int deletpoint, int newtriang, int makropoint1, int makropoint2), where deletpoint is the global number of the deleted point, makropoint1 and makropoint2 are the global points of the coarsened edge in the new triangle. This function is executed in every coarsening.

The following two functions are necessary because during the coarsening, gaps in the triangulation-arrays have to be filled by moving elements or points.

element_compress (ADAPT2D *ad2d, int old_element_number, int new_element_number).

point_compress (ADAPT2D *ad2d, int old_point_number, int new_point_number).

GRAPE(ad2d, "fit-bndfunctions-send")()

ADAPT2D *ad2d;

The boundary nodes which belong to a boundary curve g stored in the structure BND_FUNCTION can be projected using this method to their boundary curves. This is done by solving $g(\vec{p} + \tau\vec{n}) = 0.0$, where $\vec{p}$ is the point to project on the boundary curve and $\vec{n}$ is the normal of the secant $\vec{p}$ belongs to, using an Iteration in τ.

GRAPE(ad2d, "get-info")(list)

ADAPT2D *ad2d;
XLIST *list;

Create info list entries for a Adapt2d instance. There are two additional entries which do not appear in the list of the method "get-info" on Fe2d. These are the number and maximum number of boundary functions.

6.3.4 Geom2d

`Geom2d` adds further instance variables for normals, vertex and element colors, boundary curves and parameter lines to its superclass `Adapt2d`. Depending on the instance flag these new elements are used by the `Geom2d` methods, e.g. the display methods only display the surface, the boundary curves or the parameter lines if the corresponding flag is set.

This class is used by `Surface` and all its subclasses to build up complex geometries, see section 7.4.3. Some projects also provide methods for `Geom2d`, especially *Dual* (8.2.7, discrete minimisation and conjugation algorithms) and *Build* (8.2.4, creating triangulations).

```
#CLASS Geom2d : Adapt2d {
  int flag;           /* 0x1  GEOM2D_ACTIVE   don't modify if FLASE */
                      /* 0x2  GEOM2D_DISPLAY  don't draw geom if FALSE */
                      /* 0x4  GEOM2D_USERNORM don't comp. normal if TRUE */
                      /* 0x8  GEOM2D_DISP_BND don't draw bnd  if FALSE */
                      /* 0x10 GEOM2D_DISP_HOR don't draw hor  if FALSE */
                      /* 0x20 GEOM2D_DISP_VER don't draw ver  if FALSE */
  VEC3 *normal;
  VEC3 *color_vertex;
  VEC3 *color_patch;
  LIST_OF_INST *bnd_curve;
  CHAIN *hor, *ver;
};
```

6.3.4.1 Memory Methods on `Geom2d`

(GEOM2D *) GRAPE(Geom2d, "new-instance")(name)

CLASS *Geom2d
char *name

Create an instance of class `Geom2d` with name `name`. This method just allocates the structure and initializes some data, the vertex and element lists have to allocated by `"list-alloc"`. All `Geom2d` instances get a default FE function `f`, the z height, and a default `interpol` function which handles the refinement of the boundary curves.

The display methods will not draw the boundary curves and parameter lines unless the corresponding flag is set in the `Geom2d` `flag`. If the user supplies normals, vertex colors or patch colors they will be used by `"display"`, if normals are supplied additionally the `GEOM2D_USERNORM` flag has to be set.

GRAPE(geom, "free")()
GRAPE(geom, "zero-refs")()

GEOM2D *geom

`"free"` deletes an instance of class `Geom2d`, for details see 5.4.4.1. Since the boundary curves and parameter lines reference the `Geom2d` instance they are part of, a `"zero-refs"` method which determines the real number of references has to be provided.

GRAPE(geom, "skip-bnd-send")()

GEOM2D *geom

Remove all boundaries which are not `ACTIVE` from the `geom->bnd_curve` list and delete them. For example after merging two `Geom2d` instances, `"close-send"` will only deactivate the boundary lines which are identified, this method then can be used to remove them.

GRAPE(geom, "delete-lines-send")()

GEOM2D *geom

If the parameter lines `hor` and `ver` were created, for example by reading an description file for an explicit surface in project *Explicit*, but are not needed they can (and should) be deleted with this method. Especially if the surface is reflected these lines can take up a large amount of memory (or disk space if the surface is written), deleting them also makes many operations like interpolation much faster.

GRAPE(geom, "list-alloc")(points, elements)
GRAPE(geom, "list-free")()

GEOM2D *geom
int points, elements

Allocate or free memory lists for `points` points and `elements` elements for a `Geom2d` instance. Memory for normals or colors will only be allocated if the corresponding pointers are not `NULL`, i.e. if this method is used to reallocate memory (see 5.4.4.2 for general information about the `"list-alloc"` concept).

GRAPE(geom, "point-list-alloc")(points, connect)
GRAPE(geom, "point-list-free")()

GEOM2D *geom, *connect
int points

Allocate or free memory for all vertex lists, i.e. for coordinates, normals and vertex colors. If `connect` is not `NULL` its connectivity (`vertex` and `neighbour`) is also used for `geom`.

In contrast to "list-alloc" "point-list-alloc" doesn't support the reallocation concept. You should be aware than using the same connectivity arrays for more than one instance can be dangerous since these arrays don't have reference counters.

These methods are only provided for backward compatibility.

GRAPE(geom, "sort-points-send")()
GRAPE(geom, "list-shrink-send")()
GRAPE(geom, "optimize-mem-send")()

GEOM2D *geom

"sort-points-send" removes all unused points (points whose indices don't occur in the vertex list) from geom and corrects the vertex indices of boundary curves and parameter lines. This is for example useful after merging and closing several Geom2d instances to remove former boundary points which are no longer used. The method was rewritten and is much faster now, especially for large numbers of points.

"list-shrink-send" frees the unused space between number_of_points/elements and max_number_of_points/elements in the geom vertex and element lists (by calling "list-alloc" with number_of_points and number_of_elements as parameters). "list-shrink-send" is also send to all boundary curves and parameter lines.

"optimize-mem-send" combines both methods to clean up geom. First "sort-points-send" is called to remove unused points, then the memory lists are shortened with "list-shrink-send".

GRAPE(geom, "hardcopy")()
GRAPE(geom, "softcopy")(copy)

GEOM2D *geom, *copy

Create a copy of geom which its own vertex lists (all data is copied) and return the pointer to it. "hardcopy" and "softcopy" differ in two respects. While "hardcopy" creates an identical copy of geom, "softcopy" only copies the memory needed, i.e. upto number_of_points/elements. If copy is NULL, "softcopy" creates a new Geom2d instance to store the copy, otherwise copy is used.

GRAPE(geom, "softcopy-bnd-curve")(copy)
GRAPE(geom, "softcopy-lines")(copy)

GEOM2D *geom, *copy

Same as "softcopy" but only the boundary curves bnd_curve resp. the parameter lines hor and ver are copied. Of course copy has to exist for these methods.

GRAPE(geom, "copy-refine")(source)

GEOM2D *geom, *source

Copy the refine array from source to geom. This method is useful when several Geom2d instances with the same connectivity have to be refined.

6.3.4.2 Display Methods on Geom2d

GRAPE(geom, "display")()

GEOM2D *geom

Display a Geom2d instance on the standard graphicdevice as a patch or as a grid model. If geom is displayed in patch mode and normal is NULL or GEOM2D_USERNORM is not set in flag then normals are computed by calling "compute-normals-send".

The geom->flag controls which parts of geom are displayed:

GRAPE(geom, "set-geom-send")()
GRAPE(geom, "set-bnd-send")()
GRAPE(geom, "set-hor-send")()
GRAPE(geom, "set-ver-send")()
GRAPE(geom, "unset-geom-send")()
GRAPE(geom, "unset-bnd-send")()
GRAPE(geom, "unset-hor-send")()
GRAPE(geom, "unset-ver-send")()

GEOM2D *geom

Add `GEOM2D_`-flags (see the `Geom2d` structure above) to or remove them from `geom->flag`. Surface, boundary curves and horizontal or vertical parameters lines are only displayed if the corresponding flag is set.

To display boundary curves or parameter lines independent from `geom->flag` the following methods can be used:

GRAPE(geom, "bnd-disp")()
GRAPE(geom, "hor-disp")()
GRAPE(geom, "ver-disp")()

GEOM2D *geom

Display the boundary lines or the horizontal or vertical parameter lines.

GRAPE(geom, "normals-disp")()
GRAPE(geom, "normals")(length)

GEOM2D *geom
double length

Display the `geom` together with its `normals`. If the `GEOM2D_USERNORM` flag is not set the normals are computed with `"compute-normals-send"`. The normals are drawn as lines with length `length`, the interactive method `"normals-disp"` adds a ruler to the manager which allows to edit this length.

GRAPE(geom, "refine-display")()

GEOM2D *geom

Display the edges that would be refined and the edges that would be created by calling `"refine(-send)"`. This method extends the superclass method (see 6.3.3.2), it allows to display boundary curves or parameter lines.

6.3.4.3 I/O Methods on `Geom2d`

GRAPE(geom, "xdr")(xdrp)

GEOM2D *geom
XDR *xdrp

Read the instance `geom` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

6.3.4.4 Debug Methods on `Geom2d`

GRAPE(geom, "get-info")(xlist)

GEOM2D *geom
XLIST *xlist

Add information about normals, colors, boundary curves and parameter lines to the superclass information. You can get more information about the boundary curves and parameter lines by selecting them in the manager and viewing their info lists.

GRAPE(geom, "follow")(follow, pop_push)

> **GEOM2D *geom**
> **INSTANCE *follow[3][3]**
> **char pop_push[3][3]**

Define the manager environment for the `geom` instance. This will allow to interactively access the boundary curves (top right) and the horizontal and vertical parameter lines (right and bottom right).

GRAPE(geom, "show-send")()

> **GEOM2D *geom**

Print the data of a `Geom2d` instance to `stdout`.

6.3.4.5 Other Methods on `Geom2d`

GRAPE(geom, "compute-normals-send")()
GRAPE(geom, "get-vertex-wnormals-send")()

> **GEOM2D *geom**

These methods compute vertex normals for a `Geom2d` instance and store them in the `normal` array, if `normal` is `NULL` new memory is allocated. Most methods which need normals call `"compute-normals-send"` if `normal` is `NULL` or `GEOM2D_USERNORM` is not set in the `geom->flag`.

`"compute-normals-send"` computes the normal at each vertex by averaging the triangle normals of the adjacent triangles. `"get-vertex-wnormals-send"` additionally weights the triangles normals with the angle of the triangle at the vertex, this needs a little more time but the result is much better.

`"get-vertex-wnormals-send"` sets the `GEOM2D_USERNORM` flag, therefore the `geom` should not be changed after this method has been called.

GRAPE(geom, "flip-normals-send")()

> **GEOM2D *geom**

Change the orientation of the normals. This might be necessary if the orientation of the normals computed by an algorithms differs from the "natural" orientation induced by the orientation of the triangle (this is for example the case for surfaces computed with project *Amandus*).

(LIST_OF_INST *) GRAPE(geom, "compute-symmetries")()
GRAPE(scene, "compute-symmetries-send")()

> **GEOM2D *geom**
> **SCENE *scene**

`"compute-symmetries"` expects `geom` to be a part of a triply periodic surface bounded by planar symmetry lines. It returns a list of `Triang1d` instances describing the symmetry polyeder which bounds `geom`.

The methods `"compute-symmetries-send"` allows to insert such a list into the data hierarchy. The `object` of `scene` must be of class `Geom2d`, the list returned by the `"compute-symmetries"` call on the `object` is inserted as `object` of `scene->next_scene` (which is created if necessary).

GRAPE(geom, "compute-bndlen-send")()
GRAPE(geom, "compute-periods-send")()

> **GEOM2D *geom**

Prints the lenghts of the boundary curves resp. the periods (i.e. the distances between planar boundary curves whose planes are parallel) of `geom` to `stdout`.

GRAPE(geom, "equiangulate-send")()
GRAPE(geom, "equiangulate-s3-send")()

GEOM2D *geom

Equiangulate `geom` by switching diagonals between adjacent triangles if the angles opposite to the diagonal add up to more than pi (see the same methods on `Triang2d` in section 6.3.1.4). The boundary neighbours have to be corrected after this operation. The s3-method does the same operation but the points are projected to the three-sphere first.

GRAPE(geom, "mollify-send")()
(int *) GRAPE(geom, "get-mollify-bnd")()

GEOM2D *geom

Mollify `geom` by averaging the points. `"mollify-send"` extends the superclass method, if the "mollify boundary" flag (a pointer to it can be obtained by `"get-mollify-bnd"`) is `TRUE`, boundary points will also be mollified. Of course this only makes sense if boundary restrictions are available.

GRAPE(geom, "center-send")()
GRAPE(geom, "normalize-send")()

GEOM2D *geom

`"center-send"` moves `geom` to make the minimum and maximum x/y/z-coordinates identical (apart from the sign), i.e. it centers `geom` at zero. `"normalize-send"` additionally scales `geom` so that it fits into the [-1, +1] cube.

Both methods extend the superclass methods, they additionally correct the coordinates of the boundary points.

It is not recommended to use these methods to get `geom` in a better position for viewing it, for this you should use `"fit2window-send"` which changes the `GraphicDevice`'s transformation matrix instead of the coordinates of the `geom` points.

GRAPE(geom, "make-bnd-send")()

GEOM2D *geom

Compute the boundary of a `Geom2d` instance. All boundary edges are determined using the `neighbour` information, they are used to create boundary curves. If the angle between two edges at a commen vertex is too small (less than 125 degrees) this vertex is used a start resp. endpoint.

GRAPE(geom, "assign-bnd-send")()

GEOM2D *geom

Call `"assign-curve-send"` (see 6.2.2.5) on all active boundaries in `geom->bnd_curve` to update the boundary curve coordinates.

GRAPE(geom, "restrict-bnd-send")()

GEOM2D *geom

Call the restriction function for all active boundaries in `geom->bnd_curve`. This is for example done after refining `geom` to correct the coordinates of new boundary points.

GRAPE(geom, "fix-bnd-send")()

GEOM2D *geom

Remove all boundary curve restriction. Most methods respect the boundary restrictions, these methods will not move any boundary points after this methods has been called.

(int *) GRAPE(geom, "get-bnd-index")(point, dist)

GEOM2D *geom
VEC3 point
double dist

Search all active boundary curves in `geom->bnd_curve` for a point that has a distance less than `dist` from `point`. If such a boundary is found a pointer to its index is returned, otherwise the method returns `NULL`.

Normally this method is called after `"find-pick"` or `"find-pick-index"` (both methods return a point picked with the mouse) to find the boundary the picked point belongs to.

GRAPE(geom, "interpol")(geom1, geom2, factor)
GRAPE(geom, "interpol-fac")(geom1, geom2, fac1, fac2)

> **GEOM2D *geom, *geom1, *geom2**
> **double factor, fac1, fac2**

These methods extend the superclass interpolation methods, they additionally interpolate all special `Geom2d` components (normals, colors, boundary curves and parameter lines) if present.

GRAPE(geom, "merge")(geom_two)

> **GEOM2D *geom, *geom_two**

Merge `geom_two` into `geom` by appending all `geom_two` lists to those of `geom`. New memory is allocated if necessary, if something fails the method returns `NULL`.

`geom_two` should be deleted after `"merge"` because some objects of `geom_two`, e.g. the boundary curves, are assigned to `geom`, they no longer are connected to `geom_two`. This might be changed in a future version.

GRAPE(geom, "close-send")()
GRAPE(Geom2d, "close-config-send")()

> **GEOM2D *geom**
> **CLASS *Geom2d**

If `geom` consist of several separate surface patches which meet at boundary curves (e.g. if `geom` was created with the surface builder or build by reflection and `"merge"` was called) `"close-send"` identifies the double (or multiple if more than two boundaries meet) points and extends the `neighbour` connectivity over the whole surface. References within boundary curves or parameter lines to the points that are no longer used are corrected.

`"close-config-send"` pops up a configuration layer which allows to change the distance upto which points are identified, to select whether very small boundaries should be removed completly and whether some debugging information should be printed, see section 8.2.1.

Since the identified points and curves are only deactivated it might be a good idea to call `"optimize-mem-send"` after `"close-send"` to remove them.

GRAPE(geom, "reflect")(trfmat, orient)
GRAPE(geom, "reflect-non-linear")(type, points)

> **GEOM2D *geom**
> **MATRIX44 trfmat**
> **int orient**
> **int type**
> **VEC3 points[3]**

Both methods create a new `Geom2d` instance by reflecting `geom` and return it.

`"reflect"` transforms all points and normals with `trfmat`, boundary curves and parameter lines are also transformed by calling `"reflect"` on them. If `orient` is negative the triangle orientation is changed, if `g_fabs(orient) > 1` the normal orientation is also changed.

`"reflection-non-linear"` knows R^3 circle, H^3 plane, line, point and normal and S^3 plane and line reflection (`type` bitwise from 0x20 to 0x800). The `points` describe the point, line, plane or normal, for H^3 the Klein model and for S^3 the stereographic projection is used.

GRAPE(geom, "refine-local-send")()
GRAPE(geom, "coarsen-global-send")()

GEOM2D *geom

These methods extend the superclass methods "refine-local" and "coarsen-global-send". After refining geom2d "restrict-bnd-send" is called to correct new boundary points, after coarsening new boundaries are created with "make-bnd-send".

GRAPE(geom, "reset")()

GEOM2D *geom

Reset the number_of_points/elements of geom and all boundary curves and parameter lines to zero.

GRAPE(geom, "export-evolver-send")()

GEOM2D *geom

Export a Geom2d instance in Evolver format. The data is written into the file grape.fe in the current directory. Planar constraints are preserved, therefore it is possible to create initial surfaces with free planar boundaries with **GRAPE** and to minimize them with the Evolver.

6.4 Volumes and 3D Finite Elements

We have introduced the class Triang3d in order to describe the shapes of domains in the dimension three.

```
#CLASS Triang3d : Root {
        int number_of_points;
        int max_number_of_points;
        double *x, *y, *z;    /* point-coordinates */
        int number_of_elements;
        int max_number_of_elements;
        INT4 *vertex;        /* vertex indices of all tetrahedron */
        INT4 *neighbour;     /* neighbour tetrahedron */
                             /* neighbour[j][i] is opposite to vertex[j][i] */
        int data_is_borrowed;
};
```

In order to employ finite elements a subclass Fe3d of Triang3d need to be introduced.

```
#CLASS Fe3d : Triang3d {
        int dimension_of_value; /* 1==scalar,  >1 => vector function     */
        int polynomial_order;   /* 1==linear, 2==quadratic,... elements  */
        void (*f)(struct fe3d *, int, VEC4, double*);
                                /* void *f( FE3D *fe, int triangle,
                                            VEC4 barycentric_coordinates,
                                            double value[dim] )          */

        /* optional ******************************************************/
        int size_of_data;      /* size of user data                      */
        char *data;            /* and pointer to it                      */
        /* **************************************************************/
};
```

Now let us specify for each method the message passing, the delivering of arguments and the return value. We will also describe briefly the method itself. (Note: There will occur methods working on instances of Scene. At the first scene node the object is a Fe3d instance registered and the following next_scene node contains the inserted instance after the return from the actual method.)

6.4.1 Triang3d

6.4.1.1 General Methods on Triang3d

GRAPE(triang3d, "display")()

TRIANG3D *triang3d ;

The instance triang3d is rendered on the standard device in a default manner.

GRAPE(tr, "show-send")()

TRIANG3D *tr ;

The data of an instance tr is written to the standard output.

GRAPE(tr, "c-show")()

TRIANG3D *tr;

The data of an instance tr is written to the standard output. The output has the outlook of a C source file.

6.4.1.2 The Methods Read, Write and Free

(TRIANG3D *) GRAPE(Triang3d, "read")()

CLASS *Triang3d

Read a `Triang3d` instance from the standard textdevice. If successful the instance otherwise `NULL` is returned. This method is only provided for backward compatibility to read files written with `"write"`, it is replaced by

GRAPE(triang, "xdr")(xdrp)

TRIANG3D *triang
XDR *xdrp

Read the instance `triang` from or write it to the XDR stream `xdrp`. This method shouldnot be called directly, refer to 5.7 (especially 5.7.3) for details.

GRAPE(triang3d, "free")()

TRIANG3D *triang3d;

The associated memory of the instance variables is freed-up and the instance are deleted. For a more detailed description of this event see 5.4.4.1. This method is normally only called by the delete-method on class ROOT.

These three methods are also applicable to the class Fe3d.

GRAPE(fe3d, "xdr")(xdrp)

(FE3D *)GRAPE(Fe3d, "read")()

GRAPE(fe3d, "free")()

The methods for reading, xdr and deleting a Triang3d instance are extended ones because of the additional data of the subclass. The corresponding finite element function is treated as linear. After writing and reading commands have been given this situation can cause for example a change from a quadratic finite element function to a linear one. If there is no special read facility for the user's data it is possible to read the Triang3d part of an earlier written fe instance.

6.4.1.3 Special Methods on Triang3d

GRAPE(triang3d, "list_alloc")(max_points, max_elements)

GRAPE(triang3d, "list_free")()

TRIANG3D *tr;
int max_points, max_elements;

This method allocates or frees the memory of the coordinate and connectivity arrays in a given TRIANG3D instance tr. For further information about list-alloc see 5.4.4.2.

GRAPE(triang3d, "check-send")()

TRIANG3D *triang3d ;

A set of tetrahedron is tested on correct connectivities, orientation and positive volumes.

GRAPE(tr, "check-phase")(phase)

TRIANG3D *tr;
int phase;

This method invokes the checking of specific data in the geometry. There are the following possibilities at hand (a bigger number includes all the lower ones):

- Phase 0 - Global Variables
- Phase 1 - vertex field
- Phase 2 - neighbour field
- Phase 3 - connectivity
- Phase 4 - Euler-Poincare-Characteristic
  - a - full volume (tetrahedron)
  - b - boundary (triangles)
- Phase 5 - location of points with respect to common face of neighb. tetra.
- Phase 6 - positive volume

(TRIANG3D *)GRAPE(tr3, "collect")(fine)

double fine ;

Coarsens tetrahedral mesh by identifying 2 adjacent points This method performs the identification of points those distance is less than the parameter fine. All tetrahedron which are on the edge joining these points are deleted.

NOTE : If a neighbourhood-relation is present it is corrected. If a tetraedation has been extended, (by mirroring for example) tetraeders at the former boundary still have no proper relation. This means the collect operation here is a local one with no global search.

(TRIANG3D *)GRAPE(tr3d, "center")()

TRIANG3D *tr3d ;

The centerpoint of all points of the whole triangulation is set to 0.0. This helps to manage the object in the interactive environment.

(TRIANG3D *) GRAPE(t, "find-pick-intersec")(v,u,w,element,bary)

> **TRIANG3D *t;**
> **VEC3 v;**
> **double u,w ;**
> **int *element;**
> **double [4] bary;**

These methods find on the current intersection plane of the current clipping method the intersection point of the ray (u,w,.) in screen coordinates where u,w are between -1 and 1. Afterwards the coordinates of the vector, the picked element and the barycentric coordinates are v,*element and bary. The method returns the triangulation if such an intersection could be found otherwise a NIL pointer.

GRAPE(triang3d, "clip")(f, var, draw_clip, scal, clip_flag)

(TRIANG3D *)GRAPE(triang3d, "clip-ball-disp")()

(TRIANG3D *)GRAPE(triang3d, "clip-plane-disp")()

(TRIANG3D *)GRAPE(triang3d, "clip-deg-fine-disp")()

> **TRIANG3D *triang3d;**
> **double (*f)();**
> **char *var;**
> **int (*draw_clip)();**
> **char *scal;**
> **int clip_flag;**

The methods are designed to support general clipping facilities with varying hypersurface and local rendering function. A set of tetrahedron is divided by the implicitly given clipping function f into a visible ($f > 0$) and an invisible ($f < 0$) part. The zero level of this clipping function defines the clipping level.

The function f must be of the following form:

f(x,y,z,var)

> **double x,y,z ;**
> **char *var ;**

The variable var in the parameter list of the method call and of the function f must be of the same type. *var determines the clipping function. The implicit function description of a plane or a ball are predefined:

```
double plane(x,y,z,var)
      double x,y,z;
      PLANE_PARM *var;

(PLANE_PARM = { double n_x,n_y,n_z,distance;} [Hesse Form])

double ball(x,y,z,var)
      double x,y,z;
      BALL_PARM *var;

(BALL_PARM = { double center_x,center_y,center_z,radius;})
```

The pointer draw_clip points to a function describing the visualization of the surfaces of intersections. A NIL pointer causes the standard visualization of the surfaces including normal shading. The predefined function draw_clip_deg_fine (scal = clip3d_par) shows the size of each clipped tetrahedron in colour. In the interactive method clip_deg_fine_disp a plane can be varied for the clipping.

For the construction of own rendering functions for the surface of intersections we refer to the remarks on the clip3d_par structure in the description of the method "clip" on a Fe3d instance.

(TRIANG3D *)GRAPE(Triang3d,"make")(x,y,z,x_n,y_n,z_n,inst_name)

(TRIANG3D *)GRAPE(Triang3d,"make6")(x,y,z,x_n,y_n,z_n,inst_name)

double *x,*y,*z;
int x_n,y_n,z_n ;
char *inst_name ;

A Triang3d instance is built starting with an underlying rectangular mesh. The points of this mesh are (x(i),y(j),z(k)) with $(0 <= i < x\_n, 0 <= j < y\_n, 0 <= k < z\_n)$. In this way finite differences may be mapped to finite elements. In case one of the variables x, y or z is a NIL pointer a regular discretization regarding this coordinate is calculated.

The resulting mesh consists of 5 tetrahedron for each cube in the 3D mesh. The method "make6" does the same, except that 6 tetrahedron form one cube. This kind of discretization is more regular and therefore better conditions for finite element calculations.

The name of the new instance is inst_name.

(TRIANG3D *)

GRAPE(Triang3d, "mask-make")(x,y,z,x_n,y_n,z_n,mask,inst_name)

(TRIANG3D *)

GRAPE(Triang3d, "mask-make6")(x,y,z,x_n,y_n,z_n,mask,inst_name)
double *x,*y,*z;
int x_n,y_n,z_n;
int **mask;
char *inst_name;

These methods generate a Triang3d instance containing the triangulation of a cubic discretized domain. This domains consists of those cell which are marked by a YES (1) in the 3d int array mask (int mask[x_n-1][y_n-1][z_n-1]). The name of the new instance is inst_name.

GRAPE(tr,"shrink-disp")()

GRAPE(tr, "shrink-foc-tetra-disp")()

GRAPE(tr, "shrink-foc-point-disp")()

GRAPE(tr, "shrink-layer-disp")()

GRAPE(tr, "shrink-angle-disp")()

GRAPE(tr, "shrink")(p,f,fpar)

TRIANG3D *tr;
double p;
int (*f)(TRIANG3D *, int, char *); /* f(tr, i, fpar)
char *fpar;

The set of tetrahedron belonging to the Triang3d instance is displayed on the standard device as a patch model. It is possible to identify each tetrahedron, because they are separated from each other through a small space given by the parameter p. There is a selection criterion for those elements which are of interest. The specific tetrahedron is displayed if f(tr,i,fpar) is YES (1). There are several interactive versions with different functions f.

"shrink-layer-disp" displays all tetrahedron those centerpoints lie in a layer surrounding a certain plane.

"shrink-angle-disp" displays all those elements with special angles between two bounds.

"shrink-focus-point-disp" displays the neighbourhood of a certain point with an index given by a ruler variable.

The method "shrink-focus-tetra-disp" does the same for an element index.

(TRIANG3D *)GRAPE(tr3,"identify-nbs-send")()

TRIANG3D *tr3;

"identify-nbs-send" builds neighbourhood information for an arbitrary tr3 only out of tr3->vertex information. If not existent the memory for tr3->neighbour is allocated.

6.4.1.4 Extracting the Boundaries of Triang3d-Instances

GRAPE(triang3d, "select-boundary-disp")()

GRAPE(triang3d, "select-boundary")(value)

TRIANG3D *triang3d ;
int value;

Only those boundary-surfaces of a 3d-object are displayed, whose boundary value equals the parameter *value*. In the disp method the *value* can be set with a ruler.

GRAPE(triang3d, "boundary-ident-send")()

TRIANG3D *triang3d ;

One singular surface of a 3d-object can be chosen when clicking it with the left mouse button. The edges of the object separate the various surfaces. The minimal angle only just specifying an edge can be set by the root-method "choice-edge-angle-send". After clicking the middle mouse button the whole object is displayed instead of the singular surface. When clicking the right mouse button the method will be finished, the choice of the surface will be saved and a layer added:

The chosen surface becomes the negative value which is entered in the input. The Button *another selection by picking* starts the method again.

(TRIANG2D *)GRAPE(triang3d, "generate-boundary")()

(SCENE *)GRAPE(s,"generate-boundary-send")()

TRIANG3D *triang3d;
SCENE *s; /*of which the object is 3d)*/

The boundary of a 3d-object will constitute a 2d-triangulation, which in the scene method will be put into the scene tree. In most cases the following problem will arise: the triangle-normals will also be interpolated at the edges, which therefore appear blurred. The interpolation of the normals at the edges can be avoided using the triang2d-methods "show-edges-send" or "split-surfaces-send" (see 6.3.1.4).

6.4.2 Fe3d

6.4.2.1 Special Methods on Fe3d

GRAPE(fe, "point-probe-send")()

The value of a function at a position the user has to pick by mouse click is displayed.

GRAPE(fe, "clip")(f,var,draw_clip,scal,clip_flag)

GRAPE(fe, "clip-isoline-disp")()

GRAPE(fe, "clip-iso-select-disp")()

GRAPE(fe, "clip-in-out-disp")()

GRAPE(fe, "clip-vect-disp")()

GRAPE(mgr, "set-clip-parm")(nx,ny,nz,d,is,io,ih,ms,vl)

```
FE3D *fe ;
MANAGER *mgr;
SCENE *s;
double (*f)();
char *var;
int (*draw_clip)();
char *scal;
int clip_flag;
double nx,ny,nz,d,is,io,ih,ms,vl;
```

Different to the 'clip' method on a Triang3d instance the function of the Fe3d instance can now be employed to define the draw_clip function. This enforces all those intersection methods which include function depending drawing to act on the class Fe3d.

There are some predefined ones:

- **draw_clip_isoline:** produces an isoline image of a one dimensional FEM function or of the absolute value of a vector valued function.
- **draw_clip_iso_select:** displays in grid mode only those isolines whose values are stored in the array colorbar->values. In patch mode these particular values are displayed as white lines above the isoline image.
- **draw_clip_vect:** draws a mesh of vector projections, where colour represents the normal component.
- **draw_clip_in_out:** renders the stream in and stream out of the surface in form of level lines.
- **draw_clip_in_out_rel:** the length of the velocity vectors is considered additionally in the calculation of the colour in the function draw_clip_in_out.

The method set-clip-parm allows the user to predefine the frequently used variables of the interactive clipping routines. (nx,ny,nz are the components of the plane normal, d its distance from the origin, is,io the scaling and the coloroffset of the isolines,ih the graph height over the clipping plane, ms the mesh size of vector fields, and vl the vector length scaling).

The user himself can add new clipping methods by changing the hypersurface or by adding new local rendering functions. Therefore we now explain the parameter structure for the local rendering function **draw_clip**. The general 'clip' method calls this function in the following way:

draw_clip(clip3d_par)

The structure clip3d_par must be used to construct a draw_clip function:

```
typedef struct clip3d_par {
       TRIANG3D *triang3d; /* instance of class Triang3d */
       int it;             /* number of tetrahedron      */
       int n;              /* number of points of the */
                           /* intersection */
       VEC3 v[5];          /* v[0]: coordinates of visible */
                           /* tetrahedron point, v[1]..v[n]: */
                           /* vector with the coordinates of the */
                           /* intersection points */
       double bary[5][4];  /* barycentric coordinates of the */
                           /* intersection points */
       double (*f)();      /* the clipping function f(x,y,z,var) */
       char *var;          /* pointer to parameters of the */
                           /* clipping function */
       char *scal;         /* pointer to the parameters of */
                           /* draw_clip function, as they were */
                           /* passed to the original clip */
                           /* routine */
     } CLIP3D_PAR;
```

This structure describes the shape of the intersection of the clipping function level and the tetrahedron it. The filled structure is delivered by the actual clip routine.

The vector v[0] contains the coordinates of a visible tetrahedron point. v[1· · ·n] are the coordinates of those points which lie on edges of the tetrahedron it and in the surface described by the clipping function f.

In all the listed interactive methods a plane is used as the clipping function.

(TRIANG3D *)GRAPE(tr, "fix-clip")(f,fpar)

(TRIANG3D *)GRAPE(s, "fix-clip-send")()

TRIANG3D *tr;
SCENE *s;
double (*f)();
char *fpar;

This method performs the same clipping operation and builds up a TRIANG3D instance containing the sub level of this function cut with the geometry. The methods "fix-clip-send" takes the object of the scene node and the current kind of clipping from the "disp" methods and inserts a new scene node containing the clipped TRIANG3D instance as its object.

GRAPE(fe, "bnd-isoline-disp")()

GRAPE(fe, "bnd-iso-select-disp")()

FE3D *fe;

These methods produce an isoline image of a one dimensional FEM function (or of the absolute value of a vector valued function) **at the boundary** of the Fe3d-Instance. The method "bnd-iso-select-disp" displays in grid mode only those isolines whose values are stored in the array colorbar ->values. In patch mode these particular values are displayed as white lines above the isoline image.

Additionally a ruler *bnd-value (for bnd-isoline-disp)* appears to choose the *bnd-value*. If these value is less then 0, the isolines are displayed only at those boundary-faces, of which the neighbour-value is equal to *bnd-value*. If the value is 0, isolines are displayed at all boundary-faces and if the neighbourhood-relations are correct, the edges of the boundary are highlighted by additional lines. The minimal angle only just specifying an edge can be set by the root-method "choice-edge-angle-send".

GRAPE(tr, "level")(func,lev)

GRAPE(fe, "level-disp")()

(TRIANG2D *)GRAPE(tr, "fix-level")(func,lev)

(TRIANG3D *)GRAPE(tr, "fix-sub-level")(funct,lev)

(TRIANG3D *)GRAPE(tr, "fix-super-level")(funct,lev)

(SCENE *)GRAPE(scene, "fix-level-send")()

(SCENE *)GRAPE(scene, "fix-super-level-send")()

(SCENE *)GRAPE(scene, "fix-sub-level-send")()

```
TRIANG3D *tr;
FE3D *fe;
SCENE *s;
void (*funct)();
double lev;
```

The slices of tetrahedron are calculated where the one-dimensional finite element function func has the level lev. If func is NIL and tr is an instance of a subclass of Fe3d then the finite element function is taken into account. In that case you can choose first or second order approximation if the polynomial order is greater than 1. For switching there is a button in the options menu. If the dimension of value of this function is greater than 1 the absolute value of the vector defines the inspected function. In the case of a three dimensional field the components are weighted, each by a specific double (with corresponding rulers). The resulting 2D level surface consists of a polygonal approximation. The "fix"-methods perform the same level calculation for the object of the scene instance and insert as the next scene in the chain the level set as a Triang2d instance. The "super" and the "sub"-methods focus to the super of sub volumes (3D triangulations) of the function on the volume geometry.

(FE3D *) GRAPE(Fe3d, "make")

(x,y,z,x_n,y_n,z_n,values,dim,inst_name)

(FE3D *) GRAPE(Fe3d, "make6")

```
(x,y,z,x_n,y_n,z_n,values,dim,inst_name)
TRIANG *tr;
double *x,*y,*z;
int x_n,y_n,z_n;
double *values;
int dim;
char *inst_name;
```

For the corresponding Triang3d instance the method "make" is called and some additional variables for a Fe3d instance are declared. The finite element function is built with linear interpolation of the function values given at the discretization points. The function value at the discretization point index_x in the x-direction, index_y in the y-direction and index_z in the z-direction is found as value[index_z + z_n * (index_y + y_n * index_x)].

The resulting mesh consists of 5 tetrahedron for each cube in the 3D mesh. The method "make6" does the same, except that 6 tetrahedron form one cube. This kind of discretization is more regular and therefore better conditions for finite element calculations.

The name of the new instance is inst_name.

(double *) GRAPE(Fe3d, "get-gradient")()

FE3D *fe;

This methods calculates the local gradient of the current fe->f function locally on the elements. Then its recovers by weighted interpolation values on the vertices of the mesh. The locally gradient calculation is exact on polynomials of degree 2 and works on the lagrangian point of 2nd order. The method returns a double field *grad[k+3*j+3*dim*i]* of size *sizeof(double)*3*dim*mnp.*

$$f^j_{,k}(p_i) = \mathrm{grad}\,[\,k + 3\,j + 3\,dim\,i\,]$$

GRAPE(fe3d,"add-grad-send")()

VEC3 *g_fe3d_gradient(fe, i, bary, jmax, Du)

VEC3 *g_fe3d_grad(fe, i, bary, _u, jmax, Du)

FE3D *fe;
int i, jmax;
f3_func *_u; /* typedef void (f3_func)(FE3D*, int, double*, double*); */
double bary[4], Du[jmax][3];

The method add-grad-send creates the Jacobian matrix field corresponding to the fe3d-data, and stores it behind the fe3d-data. If the fe-¿data - array is already larger than usual, the method asks, if the Grad field is already there (stdout). The point wise gradients are distributed by the method calc-gradient.

g_fe3d_gradient writes the first jmax rows of the Jacobian matrix(i,bary) to the array Du. The Jacobian matrix(i,bary) is the Jacobian of the point specified by the tetrahedron number i and the barycentric coordinates bary. The function assumes that the grad-field is behind the fe-data (-->add-grad-send). Du is interpolated. Return value: the field address Du (i.e. just the argument Du).

g_fe3d_grad : like g_fe3d_gradient, but the gradient is calculated directly from the fe - function (by differentiating the quadratic interpolation of the function in the midpoints and ends of the edges), i.e. you don't have to provide the gradient field mentioned above. The function returns NULL in case of trouble (which might occur in the calculation of the reference map), otherwise Du.

6.5 General Meshes

6.5.1 Revisiting numerical data and visualization requirements

Recent numerical methods deliver a large amount of data on a lot of different mesh types. The following list shows some of these types.

- Structured or unstructured meshes (from Finite Difference or Finite Element / Volume methods) with a variety of ansatz functions on them.
- Adaptive meshes consisting of a single or of mixed element types, e. g. simplicial, prismatic, rectangular or cuboidal structure or of more general kind (CSG modeling of surfaces with smooth corners naturally leads to triangles or pentagons, e. g.).

- Conforming or non–conforming meshes, where the neighbourhood of elements across an element side is not one–to–one (for example, locally refined rectangular meshes lead naturally to non–conforming meshes).
- Meshes with parametric (curved) elements of any of the above mentioned types, with globally constant or locally changing polynomial order of parameterization.
- Hierarchically structured meshes suitable for efficient numerical solvers (which give the possibility to choose a coarse hierarchy level for interactivity of time–consuming operations).

Finite Element / Finite Volume data is usually given by coefficients to ansatz functions with local support and can easily be evaluated in **local coordinates**. Therefore a function `f` on a specific element `e` of such a mesh can easily be evaluated by a call like `f(e,c)` where `c` is the local coordinate vector. For example, for piecewise linear data on triangles, the barycentric coordinates of a point in the triangle are equal to the values of the nodal basis functions. On the other hand, a piecewise polynomial interpolation of Finite Difference data, which are given only at the nodal points of a regular grid, can easily be evaluated using local coordinates as parameters.

As we want to handle 'general' elements, we use the following notion of local coordinates that belong to an element: For each element, we specify the dimension of the local coordinate system and the coordinates of all vertices of the element in that local system.

Usually, an n–dimensional element can be parameterized by an n–dimensional local coordinate system, but there are situations where it is more intuitive or appropriate to use a higher dimensional local system. Examples of this are:

- Usage of the full $(n+1)$–dimensional barycentric coordinate system on an n–dimensional simplex. If $P_1, \ldots, P_{n+1}$ are the vertices of the simplex, there is a unique representation of every point P of the simplex as $P = \lambda_1 P_1 + \ldots + \lambda_{n+1} P_{n+1}$ with $\lambda_i \geq 0$ and $\lambda_1 + \ldots + \lambda_{n+1} = 1$.
- Smooth corners in a surface built out of rectangular patches may be naturally described as triangles with 3 local coordinates or pentagons with 5 local coordinates.

As a result, we have to permit local coordinate systems of arbitrary dimension. It is natural to assume that the underlying coordinate space is a standard Euclidian vector space. The following figure shows some examples of possible local coordinate systems for distinct elements. These elements are given as the following range of local coordinates:

tetrahedron:	$\{(\lambda_0, \lambda_1, \lambda_2, \lambda_3) : \lambda_i \geq 0 \text{ and } \sum \lambda_i = 1\}$,
prism:	$\{(\lambda_0, \lambda_1, \lambda_2, h) : 0 \leq h \leq 1,\ \lambda_i \geq 0 \text{ and } \sum \lambda_i = 1\}$,
pyramid:	$\{(\mu_1, \mu_2, h) : 0 \leq h \leq 1 \text{ and } -(1-h) \leq \mu_i \leq (1-h)\}$,
cube:	$\{(\mu_1, \mu_2, \mu_3) : 0 \leq \mu_i \leq 1\}$.

6.5.2 Description of the mesh concept

The requirements from both numerical and visualization methods can be met by the data interface, which we describe in this section. It uses a procedurally linked list of elements, where the geometric type of each element may change from element to element. No fixed set of element types is prescribed, but a geometric description is used which is able to handle all types of elements which are parameterized over a convex polyhedron.

In the following sections, we describe the underlying concepts and structures (in C–notation) in a bottom–up way:

- element type description,
- element data,
- mesh data.

We present only the structures for threedimensional meshes. The corresponding structures for twodimensional meshes are straightforward and quite similar; the main difference is, that the description of 2D elements is a lot simpler.

6.5.2.1 Description of the geometric element type

The geometry of a single element is described using a polygon–oriented boundary representation of the parameter domain. As we support only (curved) polyhedral elements, an element can be described by the set of its vertices and a description of the boundary polygons. This applies only to the local coordinates of an element; its shape in world coordinates is determined by some transformation routine.

For a threedimensional polyhedron, we specify the number of boundary polygons (faces) and for each of these polygons the number of vertices, the local vertex indices and their order (thus giving an orientation to the polygon such that the surface normal is the outer normal to the polyhedron), and the local indices of the adjacent face across each of the edges of the polygon.

For each vertex, the coordinates in the local coordinate system of the element are given by the element description. Based on these values, the visualization routines may operate in the local coordinate space. The pointer to a function `check_inside()` is provided by the element description, which checks whether a point in local coordinates is inside the element or not.

The element description is completed by pointers to transformation routines from local coordinates to world coordinates and vice versa, and routines `boundary()` and `neighbour()` which give information about the neighbourship of elements. Additionally, the transformation routine from world to local coordinates checks whether a given point is inside the element or not.

```
typedef struct
{
 int       number_of_vertices;    /* number of vertices                 */
 int       number_of_polygons;    /* number of faces                    */
 int       *polygon_length;       /* vertex counts of the faces         */
 int       **polygon_vertex;      /* face to vertex adjacency           */
 int       **polygon_neighbour;   /* face to face adjacency             */

 int       dimension_of_coord;    /* dim. of local coordinate system    */
 double    **coord;               /* local coordinates of the vertices  */
 int       parametric_degree;     /* parametric element if > 1          */
 int       (*world_to_coord)(ELEMENT3D *, float *,  double *);
 void      (*coord_to_world)(ELEMENT3D *,  double *, float *);
 int       (*check_inside)(ELEMENT3D *,  double *);

 ELEMENT3D *(*neighbour)(ELEMENT3D *, int, int,  double *, float *);
 int       (*boundary)(ELEMENT3D *, int);
} ELEMENT3D_DESCRIPTION;
```

The arrays and procedures give following information:

`polygon_length[`j`]` is the count of vertices of the j–th face ($0 \le j <$ `number_of_polygons`).

`polygon_vertex[`j`][`i`]` is the (local) index (with range $[0, \ldots,$ `number_of_vertices` $- 1]$) of the i–th vertex from the j–th face ($0 \le j <$ `number_of_polygons`, $0 \le i <$ `polygon_length[`j`]`).

`polygon_neighbour[`j`][`i`]` is the index of the adjacent face to the j–th face across the edge from vertex `polygon_vertex[`j`][`i`]` to vertex `polygon_vertex[`j`][`$i+1$`]` ($0 \le j <$ `number_of_polygons`, $0 \le i <$ `polygon_length[`j`]`).

`coord[`k`][`i`]` is the i–th local coordinate of the k–th vertex from the polyhedron ($0 \le i <$ `dimension_of_coord`, $0 \le k <$ `number_of_vertices`).

`check_inside(element3d, coord)` returns `INSIDE` (-1) if the point with local coordinates `coord` is inside of the element's parameter range. Otherwise, at least one plane of a local face separates the element and the given point. In this case the local index of such a face is returned.

`world_to_coord(element3d, xyz, coord)` transforms a point from given world coordinates `xyz[3]` to `coord[dimension_of_coords]` in the local coordinate system of `element3d`. The return value is identical to that from `check_inside()`.

`coord_to_world(element3d, coord, xyz)` transforms a point from local coordinates `coord[dimension_of_coords]` to world coordinates `xyz[3]`.

`neighbour(element3d, polygon, flag, coord, xyz)` returns a pointer to an adjacent element of `element3d` across face number `polygon`. Depending on `flag`, the routine returns the first neighbour or a next one (in case of non–conforming meshes the adjacency may be not one to one). If `coord` is not `NIL`, the routine transforms those coordinates from the old local coordinate system to the new one. In many cases this transformation can be supported by the optional parameter `xyz`, which provides world coordinates of the point if available, otherwise this parameter has to be `NIL`. The structure `element3d` may be overwritten by the routine.

`boundary(element3d, polygon)` returns 0, if `polygon` is an inner face of the mesh, or a nonzero integer, if `polygon` is part of the mesh's boundary.

The number of such element descriptions and therefore the amount of storage for them is equal to the number of different element types in one mesh. There are no copies of this necessary for each element.

In the Figures, examples of vertex and face relationships for cubes, tetrahedra, and prisms are depicted. `check_inside()` is easily implemented using the local coordinates.

The user's data structures (from a numerical method, e.g.) may be completely different to the structures described above. In case of implicitly given Finite Difference meshes, the functions `neighbour` and `boundary` are implemented using simple index arithmetic.

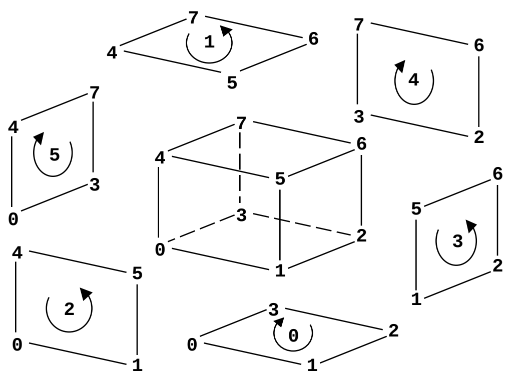


Fig. 1: Prototype of a cube

Unstructured grids usually provide such information explicitly.

Element data

We use a procedural access to single elements in this mesh concept. The procedures return element data in a small structure. This structure for an element of the mesh consists mainly of a pointer to an element description, which contains all information described above, and a vector of pointers to the world coordinates of the element's vertices.

Such information may be enlarged by optional global `eindex` and `vindex[]` integer indices of the element itself and its vertices and a pointer to a data area which contains additional information that may be used by the transformation routines or any other of the user-supplied routines.

The vertex indices, if present, allow the visualization routines to minimize the number of calls to the data function, if the data values are globally continuous. In this case, the values at a node are uniquely defined, independent of the element where it belongs to. If the data is not continuous between elements, the global vertex index may also be used to compute a continuous approximation to the data by averaging the values at the nodes.

The global element indices for example may be used to specify a set of elements of special attention and to display only these elements (or their neighbourhood).

```
typedef struct
{
  MESH3D              *mesh;        /* this is an element of 'mesh'    */
  float               **vertex;     /* world coordinates of vertices   */
  int                 *vindex;      /* indices of vertices             */
  int                 eindex;       /* index of the element            */
  ELEMENT3D_DESCRIPTION *descr;     /* connectivity, ...               */

  int                 size_of_user_data; /* opt. size of user data area*/
  void                *user_data;   /* optional pointer to user data   */
                                    /* of this element                 */
} ELEMENT3D;
```

Global access to general meshes

The mesh is just a collection of elements, as they are described in the previous section, together with an optional data function on the mesh.

We restrict the access to the elements to a procedural interface, that delivers pointers to elements in one of two possible ways:

- by building a procedurally linked list of elements, that runs sequentially through all elements in the mesh via two procedures

  `ELEMENT *first_element(mesh)`,

  `ELEMENT *next_element(element)`,

  which return the anchor to the list resp. the next element in the list.

- we can go from one element to an adjacent element (in space) across the face `polygon` via the routine

  `ELEMENT *neighbour(element, polygon, flag, coord, xyz)`.

Both the `next_element()` and `neighbour()` routines may overwrite the `element` data structure, so that no additional storage is needed for the next element's data structures. In case that one needs to collect information about several elements, a routine

`ELEMENT *copy_element(element)`

gives a copy of an element, which can be deleted later by

`free_element(element)`.

This procedural access to the mesh elements allows the generation of local element data at the time when it is actually needed. Otherwise, in case of an array or pointered list, complete information for all elements of the mesh would have to be present at the same time (and occupy much more storage).

The routines which generate the element data structures have to convert between the representations via user's data structures and the element structures. They do this mainly by filling out vertex and index information for an element and referring to a previously prepared description.

As mentioned above, in case of structured meshes this is done mainly by index arithmetic, whereas for unstructured meshes such information is usually stored with the mesh.

The `MESH` data structure contains values for the allocation of temporary memory that some routines may need:

— An upper bound `max_number_of_vertices` for the number of vertices of any element in the mesh, so that a routine which wants to store local vertex information knows how much memory it has to allocate.

— Upper bounds `max_eindex` and `max_vindex` for the element index and vertex index, if such information is supplied (then the `eindex` and `vindex[]` entries in the `ELEMENT` data structure deliver values in the range $[0, \ldots,$ `max_eindex`$-1]$ resp. $[0, \ldots,$ `max_vindex`$-1]$). If any one of the `max_?index` values is smaller or equal to zero, the elements do not give corresponding index information.

At last, the `MESH` structure contains an interface to data given *on* the mesh. Here we want to support a situation where different types of data are given at the same time on the same mesh with different value dimensions and other characteristics (example: a piecewise linear, scalar pressure, a piecewise quadratic, vector–valued velocity, and some more), together with some method to select one of the data values for display. The data characteristics may change between two visualization tasks. Additionally, there may be data characteristics which change from element to element in the same mesh, for example the polynomial degree of the basis functions. To handle all these situations, we use a rather general interface for functions on the mesh. A user supplied procedure evaluates data selected for visualization. Instead of only supporting functions given in terms of a Lagrangian basis, this concept is open for any type of user function such as one of Hermitian type or one only piecewisely defined on several parts of a single element, for example.

Data to be displayed is defined globally on the whole mesh, although the evaluation of data is done on element level (using local coordinates). Thus, the data interface is located at mesh level, and described by the following structure:

```
typedef struct
{
  char     *name;                /* textual data description        */
  int      dimension_of_value;   /* dimension of data value vector   */
  int      continuous_data;      /* data globally continuous? (0/1)  */
  void     (*f)(ELEMENT3D *, int,  double[],  double[]);
  void     (*f_el_info)(ELEMENT3D *, F_EL_INFO3D *);

  int      size_of_user_data;
  void     *user_data;

  F_DATA3D *last, *next;         /* to build doubly linked list      */
} F_DATA3D;

typedef struct
{
  int      polynomial_degree;    /* local polynomial degree of data */
} F_EL_INFO3D;
```

In detail, the entries are:

`name`: a textual description of current data (“pressure”, e.g.)

`dimension_of_value`: current dimension of data values

`continuous_data`: a flag which is 1, if the current data is continuous between elements, and 0 otherwise.

f(el, vindex, coord, val): the data function itself generating `dimension_of_value`–dimensional values at a position which has to be specified in local coordinates on element `el`. This function may also be called with parameters `coord=NULL` and a local index `vindex` of one of the element's vertices.

f_el_info(el, el_info): fills structure `el_info` with local information about data, such as polynomial degree for use by routines which adaptively choose the display resolution, e.g.

user_data, size_of_user_data: optional pointer to user's data area

last, next: to build a doubly linked list of data descriptions.

The `f_data` entry in the `mesh` structure gives access to data which is currently selected for display. By use of the doubly linked list, an interactive choice between different data is possible.

In the end, the `mesh` structure contains a data pointer `user_data` and the size of the underlying memory area `size_of_user_data`. This simplifies read and write operations and the simultaneous handling of multiple mesh3d data sets.

```
typedef struct
{
  ELEMENT3D     *(*first_element)(MESH3D *);
  ELEMENT3D     *(*next_element)(ELEMENT3D *);
  ELEMENT3D     *(*copy_element)(ELEMENT3D *);
  void          (*free_element)(ELEMENT3D *);

  int           max_dimension_of_coord;  /* max. dimension of the local   */
                                         /* coordinate system             */
  int           max_number_of_vertices;  /* max. number of vertices of    */
                                         /* one single element            */
  int           max_eindex;              /* max. global element index     */
  int           max_vindex;              /* max. global vertex index      */


  F_DATA3D      *f_data;

  int           size_of_user_data;    /* optional size of user data       */
  void          *user_data;           /* optional pointer to user data    */
} MESH3D;
```

6.5.3 Methods on Mesh3d

GRAPE(mesh, "display")()

MESH3D *mesh

The *Mesh3d* instance *mesh* is rendered on the standard device in a default manner. All the boundary faces of the elements are displayed as flat shaded patches in patch mode and in grid mode the edges of these faces are drawn.

GRAPE(mesh, "clip")(f,var,draw_clip,scal)

GRAPE(mesh, "clip-ball-disp")()

GRAPE(mesh, "clip-plane-disp")()

GRAPE(mesh, "clip-deg-fine-disp")()

GRAPE(mesh, "clip-isoline-disp")()

GRAPE(mesh, "clip-vect-disp")()

> **MESH3D *mesh**
> **double (*f)(VEC3 x,void *var)**
> **void *var**
> **int (*draw_clip)(CLIPM3D_PAR)**
> **void *scal**

These methods are designed to support general clipping facilities with varying hypersurface and local rendering function. The most basic method with the greatest flexibility is the method "clip". The others are special predefined interactive methods derived from the latter one. The domain of the mesh is split by the implicitly given clipping function f into a visible ($f > 0$) and an invisible ($f < 0$) part. The zero level of this clipping function defines the intersection surface.

The variable *var* in the parameter list of the method call and of this function f must be of the same type. The parameter **var* allows an interactive varying of the clipping function. The implicit functions describing a plane or a ball are predefined:

typedef struct MPLANE_PARM = { double n_x,n_y,n_z,distance;};

*double mplane(VEC3 x, MPLANE_PARM *var)*

typedef struct MBALL_PARM = { double center_x,center_y,center_z,radius;};

*double mball(VEC3 x, MBALL_PARM *var)*

The pointer *draw_clip* points to a function which renders the surface of intersection on a single element. This function works on the data corresponding to one single clipped element. A NIL pointer causes the standard visualization of the surfaces including normal shading.

There are some predefined local rendering functions:

- **draw_clipm_isoline:** generates an isoline image of a one dimensional FEM function or of the absolute value of a vector–valued function.
- **draw_clipm_deg_fine:** shows the size of each clipped element in colour.
- **draw_clipm_vect:** draws a mesh of vector valued function values projected onto the tangent plane of the intersection surface, where the colour represents the component normal the tangent plane (only applicable if the function is three dimensional).

The interactive methods "clip-isoline-disp", "clip-deg-fine-disp", "clip-vect-disp" use these routines while calling the standard clipping method "clip".

To construct a new private rendering method working on the surface of intersection, the user can add new clipping methods by using other and different surfaces or by adding new local rendering functions. Therefore we now explain the parameter structure for the local rendering function **draw_clip**. The general 'clip' method calls this function in the following way:

draw_clip((CLIPM3D_PAR *clip3d_par)

The structure clip3d_par with the following typedef is filled by the general "clip" method.

```
typedef struct clipm3d_par
{ ELEMENT3D *e;
  int n;
  VEC3 *v;
  double *coord;
  double (*f)();
  void *var;
  void *scal;
  int flag ;
  int dimension_of_value;
} CLIPM3D_PAR;
```

This structure describes the intersection of the hypersurface with a single element. The filled structure is delivered by the actual method "clip". The variable e points to the intersected element and n gives the number of intersection points of one connected intersection patch. The array $v[0 \cdots n-1][0 \cdots 2]$ contains the global Euclidian coordinates of these points. They are located on edges of the element's boundary at a linearly interpolated intersection position. In all the listed interactive methods a plane is used as the clipping function. *coord* is a pointer to a list of the local coordinates of the intersection points , which are stored in global coordinates in v. In this 1D array the i'th coordinate vector starts at position $(coord + i * d)$ where d is the dimension of coordinate space on the current element. The pointer *var* points to the parameters of the function describing the intersecting hypersurface.At the memory address *scal* the function *draw_clip* finds a user defined structure filled with parameters for the specific rendering style. These parameters can be influenced interactively, if the user supports this in an appropriate interactive method (see also the GRAPE reference manual).

GRAPE(mesh, "shrink")(p,f,fpar)

GRAPE(mesh, "shrink-disp")()

GRAPE(mesh, "shrink-focus-element-disp")()

GRAPE(mesh, "shrink-layer-disp")()

MESH3D *mesh
double p
int (*f)(ELEMENT3D *el,void *fpar)
void *fpar

The set of elements belonging to the Mesh3d instance is displayed on the standard device as a patch or grid model. They are separated from each other by a small gap of relative width p, this makes it possible to identify each element. There is a selection criterion for those elements which are of interest. The main method is called "shrink". Here a specific element is displayed if *f(ELEMENT3D *el,void *fpar)* is true otherwise it is skipped. There are several interactive versions with different predefined functions *f*.

- "shrink-disp" displays all elements.
- "shrink-layer-disp" displays all elements those center of mass of vertices lies in a layer surrounding a certain plane.
- "shrink-focus-element-disp" displays the neighbourhood of a certain elements. The index of this element and the depth of neighbourhood are accessible by integer rulers.

GRAPE(mesh, "level")(func,lev)

GRAPE(mesh, "level-disp")()

GRAPE(mesh, "sub-level-disp")()

GRAPE(mesh, "super-level-disp")()

MESH3D *mesh
void (*func)(ELEMENT3D *el, double *coord, double *val)
double lev

The slices of those elements where the one-dimensional finite element function *func* has the level *lev* are calculated and displayed. The paramters of *func* are equal to those used for a one dimensional FEM function on a *Mesh3d* instance: *f(ELEMENT3D *el, double *coord, double *val)*.

If *func* is NIL and *mesh* includes a function then this finite element function is taken into account. If its dimension of value is greater than 1 the absolute value of the value vector defines the function under consideration. The resulting 2D level surface consists of a polygonal approximation. The "super" and the "sub"-methods display the super or sub volumes of the function on the volume geometry.

6.5.4 Conversion Methods

The major purpose of the *Mesh3d* concept is to allow the access to data of various formats in a unique way. Therefore there is not one single format to read or write this data. It highly depends on the numerical method and the users data structures. Nevertheless some data formats will be supported. They will be discussed in the following.

(MESH3D *)GRAPE(tr3d, "convert-to-mesh")()

(MESH3D *)GRAPE(fe3d, "convert-to-mesh")()

TRIANG3D *tr3d
FE3D *fe3d

A *Triang3d* or *Fe3d* object is referred to by a *Mesh3d* instance which is the returned value of the first two methods. The original triangulation and data are kept unchanged. The *Mesh3d* evaluation functions only refer to them. Later changes in *tr3d* or *fe3d* will therefore also effect the new *Mesh3d* instance. The methods "convert-to-mesh-send" on *Scene* replaces the object in the instance *scene* by a *Mesh3d* object referring to the previous object data.

(MESH3D *)GRAPE(Mesh3d, "read-ascii-fdiff")()

(MESH3D *)GRAPE(Mesh3d, "read-binary-fdiff")()

(MESH3D *)GRAPE(Mesh3d, "write-binary-fdiff")()

These three methods support the reading and writing of finite difference data on a possibly deformed cubic mesh. The values of several solution functions can be handled at the same time. A button in the interactive environment deals with the switching between them. An automatic compression of large data sets to a smaller space resolution is possible. A ruler effects the rate of discretizations points which will be taken into account in each coordinate direction.

Remark : The *coord_to_world* function works correctly if and only if the deformation of the standard reference cube onto the deformed cube has a constant normal direction in two directions and a varying one solely in the third direction. For arbitrary deformations the user has to exchange this function by an appropriate one.

We will now list the input format for the read method dealing with the ASCII data type (the parentheses at the end of the lines contain comments) :

```
time

geo           (flag == 1 x_i y_i z_i are defined, == 0 not defined)
dim           (dimension of the solution vector)
parts         (number of segments in the solution vector,
              for example velocity, pressure, density, grad pressure ...)
part_0 name_0 (index to segment 0 of the function vector and its name)
.
.
.
part_(parts-1) name_(parts-1) (index of the last segment and its name)
```

```
n_x n_y n_z  (int number of steps in x,y,z direction)
             (n = n_x * n_y * n_z)
x_0 y_0 z_0 f_0[0]..f_0[dim-1] (x_i y_i z_i only if geo == 1)
     .                         (x is the slowest running index)
     .                         (z the fastest)
     .                         (point[i,j,k]=(x,y,z)[(i*y_n+j)*z_n+k])
     .                         (f_. solution vector on the points)
x_(n-1) y_(n-1) z_(n-1) f_(n-1)[0]..f_(n-1)[dim-1]
```

The following example illustrates this. A small 3D mesh is defined with a velocity and pressure defined on it.

```
0.0
1

4
2

0 velocity
3 pressure

3 3 3

0.0 0.0 0.0  1.0 2.0 1.0 0.012
0.0 0.0 0.5  1.0 2.0 1.0 0.012
0.0 0.0 1.0  1.2 1.6 0.5 0.018
0.0 0.5 0.0  1.2 1.6 0.5 0.018
0.0 0.5 0.5  0.9 2.2 0.7 0.008
0.0 0.5 1.0  1.2 1.6 0.5 0.018
0.0 1.0 0.0  1.0 2.0 1.0 0.012
0.0 1.0 0.5  1.2 1.6 0.5 0.018
0.0 1.0 1.0  0.9 2.2 0.7 0.008
0.5 0.0 0.0  1.2 1.6 0.5 0.018
0.5 0.0 0.5  1.0 2.0 1.0 0.012
0.5 0.0 1.0  1.2 1.6 0.5 0.018
0.5 0.5 0.0  0.9 2.2 0.7 0.008
0.5 0.5 0.5  1.0 2.0 1.0 0.012
0.5 0.5 1.0  0.9 2.2 0.7 0.008
0.5 1.0 0.0  1.2 1.6 0.5 0.018
0.5 1.0 0.5  1.0 2.0 1.0 0.012
0.5 1.0 1.0  0.9 2.2 0.7 0.008
1.0 0.0 0.0  1.0 2.0 1.0 0.012
1.0 0.0 0.5  0.9 2.2 0.7 0.008
1.0 0.0 1.0  1.2 1.6 0.5 0.018
1.0 0.5 0.0  0.9 2.2 0.7 0.008
1.0 0.5 0.5  1.0 2.0 1.0 0.012
1.0 0.5 1.0  0.9 2.2 0.7 0.008
1.0 1.0 0.0  1.0 2.0 1.0 0.012
1.0 1.0 0.5  1.0 2.0 1.0 0.012
1.0 1.0 1.0  1.0 2.0 1.0 0.012
```

If possible, the following binary format is preferable. It consists in replacing above all ASCII written integers and floats to binary written once and the character strings by char arrays of fixed length 100. Furthermore, the block of function data information is separated from the coordinate arrays.

```
(float ) time

(int ) geo,dim,parts
(int ) part_0,(char[100] ) name_0
...
(int ) part_(parts-1),(char[100]) name_(parts-1)
(int ) n_x,n_y,n_z
(float ) x_0,y_0,z_0
...
(float ) x_(n-1),y_(n-1),z_(n-1)
(float ) f_0[0], .. f_0[dim-1]
...
(float ) f_(n-1)[0], .. f_(n-1)[dim-1]
```

6.5.5 Examples for the implementation of element types

In this section we give some examples for the `ELEMENT3D_DESCRIPTION`. We point out one possible way how to define parts of the `ELEMENT3D_DESCRIPTION` for a cube, a prism, and a tetrahedron. For all elements we define `local_coord`, `polygon_length`, `polygon_vertex`, and `polygon_neighbour`. In the case of a cuboidal grid we define the main part of the function `neighbour()`. The second function `boundary()` uses the same ideas as `neighbour()` but is much easier to implement.

As a first example we start with one of the simplest mesh types and consider a set of equidistributed points given by a Finite Difference method. This should underline that beside general meshes containing various types of elements also the most basic grids can easily be handled by this concept. The whole grid is totally described by the constant mesh sizes `h_x`, `h_y`, `h_z` and the numbers of mesh points `n_x`, `n_y`, `n_z` in the x, y, and z–direction. Such information can be handled by the following

```
typedef
{
  int    n_x, n_y, n_z;     /* number of points in x, y, z-direction     */
  float  h_x, h_y, h_z;     /* distance of points in x, y, z-direction   */
  float  **f_data;          /* pointer to data matrix                    */
} USER_DATA;
```

This set of points describes a cuboidal grid. All cubes of the grid are of equal shape with diameters `h_x`, `h_y`, `h_z`. A single cube is totally described by the index triple of one vertex, for example the vertex with smallest x, y and z coordinates.

```
typedef
{
  int i_x, i_y, i_z;        /* user's reference for the actual cube      */
} CUBE;
```

For a cube with index `(i_x, i_y, i_z)` the world coordinates of the reference vertex are simple `(i_x*h_x, i_y*h_y, i_z*h_z)`. The world coordinates of the other vertices are calculated in a similar way. This is done by a procedure `fill_points(ELEMENT3D *el)`.

We use the following components of the `ELEMENT3D_DESCRIPTION` for one single cube :

```
/* vertex indices of the polygons                                       */
static int c_v0[4] = {0,3,2,1}, c_v1[4] = {4,5,6,7}, c_v2[4] = {0,1,5,4};
static int c_v3[4] = {1,2,6,5}, c_v4[4] = {2,3,7,6}, c_v5[4] = {0,4,7,3};
/* polygon adjacencies                                                  */
static int c_p0[4] = {5,4,3,2}, c_p1[4] = {2,3,4,5}, c_p2[4] = {0,3,1,5};
static int c_p3[4] = {0,4,1,2}, c_p4[4] = {0,5,1,3}, c_p5[4] = {2,1,4,0};
/* local coordinates of the vertices                                    */
```

```
static  double c_c0[3] = {0.,0.,0.}, c_c1[3] = {1.,0.,0.};
static  double c_c2[3] = {1.,1.,0.}, c_c3[3] = {0.,1.,0.};
static  double c_c4[3] = {0.,0.,1.}, c_c5[3] = {1.,0.,1.};
static  double c_c6[3] = {1.,1.,1.}, c_c7[3] = {0.,1.,1.};

static double *cube_coord[8] = {c_c0,c_c1,c_c2,c_c3,c_c4,c_c5,c_c6,c_c7};
static int   cube_polygon_length[6] = {4,4,4,4,4,4};
static int   *cube_vertex[6] =        {c_v0,c_v1,c_v2,c_v3,c_v4,c_v5};
static int   *cube_next_polygon[6] =  {c_p0,c_p1,c_p2,c_p3,c_p4,c_p5};
```

The procedure which gives adjacency information on the grid can be implemented in the following way:

```
static ELEMENT3D  *cube_neighbour(ELEMENT3D *el, int pn, int flag,
                                  double *coord, float *xyz)
{
  CUBE      *cd;
  USER_DATA *ud;

  cd = (CUBE*)el->user_data;     /* triple index of the actual element */
  ud = (USER_DATA*)el->mesh->user_data;   /* user data         */

  switch(pn) {
    case 0 :
      if (cd->i_z == 0) return(NIL);
      cd->i_z--;
      fill_points(el);
      coord[2] = 1.0;
      return(el);
    case 1:
      if (cd->i_z == ud->n_z-2) return(NIL);
      cd->i_z++;
      fill_points(el);
      coord[2] = 0.0;
      return(el);
    case 2:
      ...
  }
}
```

The description of a cube then simply is:

```
static ELEMENT3D_DESCRIPTION cube_descr =
{
  8 , 6 , cube_polygon_length, cube_vertex, cube_next_polygon,
  3, cube_coord, 1,
  cube_world_to_coord, cube_coord_to_world, cube_check_inside,
  cube_neighbour, cube_boundary
};
```

The basic functions of `MESH3D` are `first_element()` and `next_element()`. The first one has to allocate memory for one single element and for the triple index of the "user's element".

```
static ELEMENT3D *cube_first_element(MESH3D *mesh)
{
  ELEMENT3D  *el;
  CUBE       *cd;

  if (mesh == NIL)  return(NIL);
  if ((el = (ELEMENT3D*)malloc(sizeof(ELEMENT3D))) == NIL)  return(NIL);
```

```
  el->mesh = mesh;
  el->vertex = (float**)malloc(8*sizeof(float*));
  el->vindex = (int*)malloc(8*sizeof(int));
  el->user_data = (void*)(cd = (CUBE*)malloc(sizeof(CUBE)));
  el->size_of_user_data = sizeof(CUBE);
  el->descr = &cube_descr;

  cd->i_x = cd->i_y = cd->i_z = 0;
  fill_points(el);
  return(el);
}

static ELEMENT3D *cube_next_element(ELEMENT3D *el)
{
  CUBE      *cd;
  USER_DATA *ud;
  if (el == NIL) return(NIL);
  cd = (CUBE*)el->user_data;
  ud = (USER_DATA*)el->mesh->user_data;

  if(++cd->i_x == ud->n_x-1) {
    cd->i_x = 0;
    if(++cd->i_y == ud->n_y-1) {
      cd->i_y = 0;
      if(++cd->i_z == ud->n_z-1) {
        cube_free_element(el);   /* el was last element of the grid       */
        return(NIL);
      }
    }
  }
  fill_points(el);
  return(el);
}
```

For the visualization of given data from a numerical method we have to implement a procedure `f()` with corresponding `f_el_info()` in an `F_DATA3D` structure. Usually, Finite Difference methods only give values at the nodes of the grid. Such data can be interpolated by a piecewise trilinear function. In the concept of local coordinates it is very easy to evaluate such a function. Let `u[0]`,...,`u[7]` be those values at the vertices of one cube of the grid (for the sake of simplicity we assume that these are scalar values). The main part of the function `f(ELEMENT3D *el, int ind, double coord[], double val[])` would be

```
if (coord == NIL)
  *val = u[ind];
else {
  *val = (1-coord[2]) * ((1-coord[1])*((1-coord[0])*u[0]+coord[0]*u[1])+
                           coord[1]*((1-coord[0])*u[3]+coord[0]*u[2]));
  *val +=   coord[2]  * ((1-coord[1])*((1-coord[0])*u[4]+coord[0]*u[5])+
                           coord[1]*((1-coord[0])*u[7]+coord[0]*u[6]));
}
```

The values of `u[0]`,...,`u[7]` can easily be obtained from the user arrays (possibly stored at `el->mesh->user_data`) and some index arithmetic.

In the second example we give definitions for `local_coord`, `polygon_length`, `polygon_vertex`, and `polygon_neighbour` for tetrahedral and prismatic elements. Since usually these elements occur in methods using unstructured grids the adjacency connectivity of the elements usually can not be computed by indexing operations as described above. But for the Finite Element / Finite Volume methods itself the underlying user data structure usually provides such information. Thus the functions **`*neighbour()`** and **`*boundary()`** can be implemented very easily using the user's data structure.

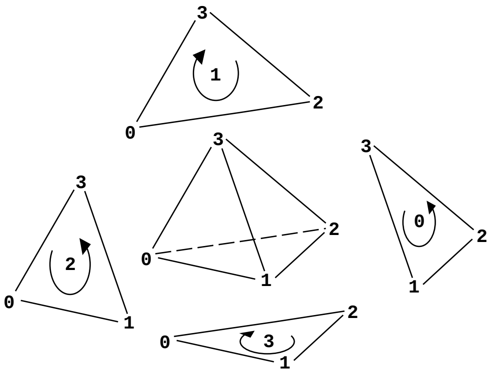
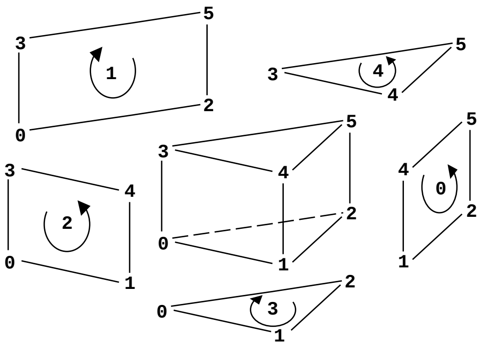


Fig. 2: Prototype of a tetrahedron



Prototype of a prism

```
/* vertex indices of the polygons for a prism and a tetrahedron          */
static int p_v0[4] = {1,2,5,4}, p_v1[4] = {0,3,5,2}, p_v2[4] = {0,1,4,3};
static int p_v3[3] = {0,2,1},   p_v4[3] = {3,4,5};
static int t_v0[3] = {1,2,3},   t_v1[3] = {2,0,3};
static int t_v2[3] = {0,1,3},   t_v3[3] = {2,1,0};
/* polygon adjacencies  for a prism and a tetrahedron                    */
static int p_p0[4] = {3,1,4,2}, p_p1[4] = {2,4,0,3}, p_p2[4] = {3,0,4,1};
static int p_p3[3] = {1,0,2},   p_p4[3] = {2,0,1};
static int t_p0[3] = {3,1,2},   t_p1[3] = {3,2,0};
static int t_p2[3] = {3,0,1},   t_p3[3] = {0,2,1};
/* local coordinates of the vertices for a prism and a tetrahedron       */
static  double p_c0[4] = {1.,0.,0.,0.}, p_c1[4] = {0.,1.,0.,0.};
static  double p_c2[4] = {0.,0.,1.,0.}, p_c3[4] = {1.,0.,0.,1.};
static  double p_c4[4] = {0.,1.,0.,1.}, p_c5[4] = {0.,0.,1.,1.};
static  double t_c0[4] = {1.,0.,0.,0.}, t_c1[4] = {0.,1.,0.,0.};
static  double t_c2[4] = {0.,0.,1.,0.}, t_c3[4] = {0.,0.,0.,1.};

static int   prism_polygon_length[5] = {4,4,4,3,3};
static int   *prism_vertex[5] =       {p_v0,p_v1,p_v2,p_v3,p_v4};
static int   *prism_next_polygon[5] = {p_p0,p_p1,p_p2,p_p3,p_p4,};
static double *prism_coord[6] =        {p_c0,p_c1,p_c2,p_c3,p_c4,p_c5};

static int   tetra_polygon_length[4] = {3, 3, 3, 3};
static int   *tetra_vertex[4] =       {t_v0,t_v1,t_v2,t_v3};
static int   *tetra_next_polygon[4] = {t_p0,t_p1,t_p2,t_p3};
static double *tetra_coord[4] =        {t_c0,t_c1,t_c2,t_c3};
```

The **`ELEMENT3D_DESCRIPTION`** for a prism and a tetrahedron then would be

```
static ELEMENT3D_DESCRIPTION prism_descr =
{
  6 , 5 , prism_polygon_length, prism_vertex, prism_next_polygon,
  4, prism_coord, 1,
  prism_world_to_coord, prism_coord_to_world, prism_check_inside,
  prism_neighbour,  prism_boundary
};
```

```
static ELEMENT3D_DESCRIPTION tetra_descr =
{
  4 , 4 , tetra_polygon_length, tetra_vertex, tetra_next_polygon,
  4, tetra_coord, 1,
  tetra_world_to_coord, tetra_coord_to_world, tetra_check_inside,
  tetra_neighbour,  tetra_boundary
};
```

6.5.6 Mesh2D

In the above discussion we always have considered the three dimensional case and argued that it is of more interest. Discussing the 2D structures is more or less straightforward. Here we present now the appropriate 2D structures including the class *Mesh2d.* The meaning of the structure entries are analogous to those already discussed in 3D. The main difference is in `ELEMENT2D_DESCRIPTION`, where everything is much simpler, because the reference element is always a convex (planar) polygon in parameter space. Thus, an array of vertex coordinates contains all geometric information. For the `neighbour()` and `boundary()` functions, edge n lies between vertex n and vertex $mod(n+1, number\_of\_vertices)$.

```
typedef struct element2d_description  ELEMENT2D_DESCRIPTION;
typedef struct element2d              ELEMENT2D;
typedef struct mesh2d                 MESH2D;
typedef struct f_data2d               F_DATA2D;
typedef struct f_el_info2d            F_EL_INFO2D;

struct element2d_description
{
  int       number_of_vertices;

  int       dimension_of_coord;
  double    **coord;
  int       parametric_degree;
  int       (*world_to_coord)(ELEMENT2D *,float *, double *);
  void      (*coord_to_world)(ELEMENT2D *,double *, float *);
  int       (*check_inside)(ELEMENT2D *, double *);

  ELEMENT2D *(*neighbour)(ELEMENT2D *, int, int,  double *, float *);
  int       (*boundary)(ELEMENT2D *, int);
};

struct element2d
{
  MESH2D                *mesh;
  float                 **vertex;
  int                   *vindex;
  int                   eindex;
  ELEMENT2D_DESCRIPTION *descr;

  int                   size_of_user_data;
  void                  *user_data;
};

struct mesh2d
{
  ELEMENT2D     *(*first_element)(MESH2D *);
```

```
  ELEMENT2D     *(*next_element)(ELEMENT2D *);
  ELEMENT2D     *(*copy_element)(ELEMENT2D *);
  void          (*free_element)(ELEMENT2D *);

  int           dimension_of_world;
  int           max_dimension_of_coord;
  int           max_eindex;
  int           max_vindex;
  int           max_number_of_vertices;

  F_DATA2D      *f_data;

  int           size_of_user_data;
  void          *user_data;
};

struct f_data2d
{
  char     *name;
  int      dimension_of_value;
  int      continuous_data;
  void     (*f)(ELEMENT2D *, int,  double[],  double[]);
  void     (*f_el_info)(ELEMENT2D *, F_EL_INFO2D *);

  int      size_of_user_data;
  void     *user_data;

  F_DATA2D *last, *next;
};

struct f_el_info2d
{
  int           polynomial_degree;
};
```

6.5.7 Methods on Mesh2d

GRAPE(mesh, "display")()

MESH2D *mesh

The *Mesh2d* instance *mesh* is rendered on the standard device in a default manner.

GRAPE(mesh,"isoline")(select,f_data,linescal,colorscal,offsetscal,highscal,max,min)

GRAPE(mesh, "isoline-disp")()

GRAPE(mesh, "isoline-select-disp")()

MESH2D *mesh
int select /* = 1 for `select` method, else 0 */
F_DATA2D *f_data
double linescal,colorscal,offsetscal,highscal,max,min

These methods display isolines (in grid mode) or iso-colored patches (in patch mode). Using the `-select` method, only specified isolines are drawn (compare the `Colorbar` documentation, Section

6.6). All parameters are similar to the `Fe2d isoline` method; `f_data` defines a function to be displayed; if this pointer is NIL, the current data function from `mesh` is displayed.

GRAPE(mesh, "value-min-max")(min, max)

GRAPE(mesh, "value-min-max-send")()

MESH2D *mesh
double *min, *max

These methods compute minimal and maximal values of the current data function on the mesh's vertices (may be used to set a Colorbar range, e.g.). The `-send` method prints the values to stdout.

6.6 Using the colorbar

The author of this section did not manage to update the following text. Refer to `grape.h` for the correct structures and types.

All display methods use various colors to visualize the data according to the evaluated function with isolines (grid mode) or colored patches (patch mode). Since the colors are not to be chosen arbitrarily, a scale is used showing the relation of colors and values. The **hsv** color model is used in GRAPE to determine the color in the display methods on an fe2d or an fe3d structure.

This color model is organized as follows:

> *Let C be the set of all available colors except black. Then the user gets a unique mapping between the data to visualize and the color, if the data values are between 0 and 1.*

Therefore it is necessary to map the range of data onto the interval [0,1]. This will be done with help of the colorbar structure if the minimum is stored in *colorbar*-->*min* and the maximum in *colorbar*-->*max*.

Those values which the lines are drawn for (on colorbar and for the isoline-methods), are calculated by the formula

$$\text{value} = colorbar\text{->}min + (n + 0.1) * \text{line-distance} \qquad \text{with } n \in \{0, 1, 2, \ldots\}.$$

The display methods using the colorbar do not determine the minimum and maximum value, the values must be provided by the user. If various display methods use different colorbars, then the colorbars are organized in a list. In this list the colorbar for each display method will be identified by the pointer of the calling method.

If particular values are of interest, they can be put onto the colorbar additionally. A name which can comprehend 100 characters at most belongs to each colorbar. However, only the first seven characters appear on a button or a ruler belonging to this colorbar. A colorbar can not be rotated.

If a colorbar is called by a GRAPE-display-method, three buttons appear automatically: a button to switch the colorbar on or off, a button to modify the colorbar and a button to change the particular values.

6.6.1 The structure of colorbar

```
typedef struct colorbar {
    /* SYSTEM ONLY */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* USER */
```

```
    INSTANCE (*funct)();        /* pointer to display method to identify
                                   the colorbar to this method           */
    INSTANCE *object;
    char name_of_function[100]; /* first 7 characters are used for the
                                   description of buttons and rulers to
                                   modify a colorbar                     */
    VEC3 xyz;                   /* position of the colorbar              */
    float height, width;        /* height and width of the colorbar      */
    int display_on_off;         /* switch to determine the visualisation
                                   of the colorbar                       */
    float eps;                  /* distance of the isolines of the colorb.
                                   in grid mode (connected to the dist.
                                   distance of the isolines of the evalu-
                                   ation of the data)                    */
    float offset;               /* offset in the hsv color model         */
    float color;                /* to scale the domain of the color      */
    float min, max;             /* minimum and maximum values of the
                                   data if set (standard is 0 and 1)     */
    int number_of_values;       /* number of values to label the colorb. */
    float values[100];          /* values to label the colorbar          */
    float input_value;          /* chosen value of the INPUT             */
    char float_display[10];     /* formatstring for value-display
                                   (standard is "%4.3f")                  */
    PURELIST *value_list;       /* list of values (in alphabet. order)   */
    SCROLLBAR *scrollbar;       /* scrollbar for these list              */
    BUTTON *button;             /* button to switch between on/off       */
    BUTTON *button_options;     /* button to get the LAYER layer_options */
    BUTTON *button_values;      /* button to get the LAYER layer_values  */
    BUTTON *button_delete;      /* button to delete values               */
    BUTTON *button_change;      /* button to change values               */
    BUTTON *button_dism_val;    /* button to delete the LAYER
                                   layer_values                          */
    LAYER  *layer_options;      /* layer to change the active colorbar
                                   by many rulers                        */
    LAYER  *layer_values;       /* layer to change the values            */
    INPUT  *input_change;       /* input in layer_values                 */
    char name_of_button[16];    /* name to label button                  */
    char name_of_button_options[16];    /* name to label button_options  */
    char name_of_button_values[18];     /* name to label button_values   */
} COLORBAR;
```

6.6.2 Methods on colorbar

(COLORBAR *)GRAPE(colorbar,"display")()

COLORBAR *colorbar

The actual colorbar will be displayed on the graphic device.

(COLORBAR *)GRAPE(Colorbar,"get-stdcolorbar")(funct,name)

CLASS *Colorbar
INSTANCE (*funct)()
char name[100]

If *funct* is equal to NIL then a default colorbar will be returned. If a colorbar with the function pointer *funct* exists this will be returned.

If in the list of the colorbars no element with function entry *funct* exists a default colorbar will be created containing *funct* and *name*. This newly created colorbar will be added to the list. *funct* is the method which calls this "get-stdcolorbar" method.

This method call for example can be found for instance within the method "fe2d-isoline-disp":

```
colorbar = (COLORBAR *)GRAPE(Colorbar,"get-stdcolorbar")
                              (fe2d_isoline_disp,"iso2dclb");
```

Furthermore the colorbar is available in the display methods "fe2d_isoline_select_disp", "fe3d_clip_isoline_disp", "fe3d_clip_iso_select_disp", "fe3d_clip_deg_fine_disp", "fe3d_bnd_isoline_disp", "fe3d_bnd_iso_select_disp" and "mesh3d_clip_isoline_disp". Note that the select-methods are using the colorbars of the correspondent isoline-methods.

(COLORBAR *)GRAPE(Colorbar,"set-default")(xyz,height,width,

display_on_off,min,max,number_of_values,values,funct,name)
CLASS *Colorbar
VEC3 xyz
double height, width
int display_on_off
double min, max
int number_of_values
float values[100]
INSTANCE (*funct)()
char name[100]

With *xyz* (coordinates of the left lower corner of the colorbar), *height* and *width* any position and size of the colorbar can be chosen. Depending on the choice of 1 or 0 for *display_on_off*, the colorbar will be displayed or not if *funct* is an actual display method. With *min* and *max* the user can specify the minimal and maximal values of the data to be visualized. In addition particularly interesting values between *min* and *max* can be marked on the colorbar (this is important for the "select"-methods, see above). *number_of_values* is the number (limited to 100) of these values. *values* is the array into which the values are written.

For example if the colorbar of the method fe2d_isoline_disp should become specifically parameters, you can add into your `main()`:

```
extern FE2D *fe2d_isoline_disp();
VEC3        xyz = {0.5, -0.9, 0.0};

GRAPE(Colorbar, "set-default")(xyz, 0.4 , 0.1, 1, -0.5, 0.5, 0, NIL,
                                fe2d_isoline_disp, "iso2dclb");
```

Often one of the shorter calls is sufficient:

(COLORBAR *)GRAPE(Colorbar,"set-default-min-max")

(min,max,funct,name)
CLASS *Colorbar
double min, max
INSTANCE (*funct)()
char *name

(COLORBAR *)GRAPE(Colorbar,"set-default-values")

(number_of_values,values,funct,name)
CLASS *Colorbar
int number_of_values
float values[100]
INSTANCE (*funct)()
char *name

(COLORBAR *)GRAPE(colorbar,"on-off")()

COLORBAR *colorbar

This call effects a switch of *colorbar-->display_on_off*.

(COLORBAR *)GRAPE(colorbar,"button")()

COLORBAR *colorbar

A button appears to switch the colorbar on or off. If the colorbar is switched on, two other buttons are shown up. By pressing them the following methods "modify" or "change-values" are sent:

(COLORBAR *)GRAPE(colorbar,"modify")()

COLORBAR *colorbar

Rulers are displayed in a layer to modify the values of *xyz, height, width, min* and *max* values of the structure colorbar.

(COLORBAR *)GRAPE(colorbar,"change-values")()

COLORBAR *colorbar

A layer appears to change the entry's of the array *colorbar->values* by buttons and inputs.

Chapter 7

Supervising Classes

7.1 The Data Hierarchy Tree

The geometric data is organized in a tree and every object belongs to one node of this tree. The simplest and mainly used nodes are of class `Scene`. They contain a pointer to two further nodes. Usually one of this nodes is a geometric object but it may also be another branch of the tree.

A subclass of `Scene` is `Chain`. `Scene` is mainly used as a binary node whereas nodes of class `Chain` are used to store a linear sequence of objects as they arise for example when generating multiple copies of an object during reflection operations. Methods on the class `Chain` assume to find such a linear structure.

A further subclass of `Scene` is `TimeScene`. A node in the hierarchy of class `TimeScene` has additionally a local time and a pointer to a dynamic object. The object of the `TimeScene` is always a section at the current local time of the dynamic process. This is the simplest way to included dynamic objects into the hierarchy.

Further hierarchy nodes can be instances of class `Time-Object`, `Curve`,`Surface` etc., see 8, for a description of these classes.

The user may also allow his own classes to appear as hierarchy nodes with objects as leaves by supplying a method "follow" for his class defining the nodes' environment.

7.1.1 Scenes — A Collection of Geometry Objects

The class `Scene` can be used to manage a collection of geometry objects. Every geometry object, e.g. any triang or finite element (fe) instance, may belong to an instance of class `Scene`. This connection is implemented by an instance pointer object. The list of scenes are chained by the pointer next-scene. The whole structure looks like a binary tree. Each scene and geometry object belongs to one node.

The properties of one scene have only an effect to the corresponding object and not to the next scene. These properties are a transformation matrix, a display method and surface properties.

Depending on the matrix-flag the given matrix object-trans will be premultiplied to the corresponding global transformation matrix. For a proper usage of the objects it is necessary to implement the methods which are demanded by the scene's method pointer method-name. If this pointer is not equal to NULL the named method is executed instead of the common `"display"` method. The available surface properties are explained in the last typedef.

These are the structures belonging to class `Scene`:

```
typedef struct scene {
    ...
    /* superclass structure */
    struct scene *next_scene;
    INSTANCE *object;
```

```
    MATRIX44 object_trans;     /* matrix to transform the objects   */
    int matrix_flag;           /* NULL (id matrix), G_MATRIX_PROJECT,
                                  G_MATRIX_VIEW, G_MATRIX_MODEL    */
    char *method_name;         /* method to use instead of display */
    SUPROP *suprop;
} SCENE;


typedef double MATRIX44[4][4];


typedef struct suprop {
    ...
    /* superclass structure */
    SUPROP_DEV suprop_dev;
    int grid_patch;
} SUPROP;


typedef struct suprop_dev {
    double emission[3];         /* rgb color emission (incl. intensity) */
    double ambient[3];          /* ambient rgb reflectance */
    double diffuse[3];          /* diffuse rgb reflectance */
    double specular[3];         /* specular rgb reflectance */
    double specular_exp;        /* specular reflectance exponent */
    double transparency;        /* transparency factor [0..1] */
} SUPROP_DEV;
```

7.1.1.1 Methods on Scene

(SCENE *)GRAPE(Scene, "new-instance")(name)

CLASS *Scene
char *name

This method creates a new instance of class `Scene` with name **"name"** and allocates the necessary memory. The transformation matrix is set to the identity matrix.

GRAPE(scene, "free")()

SCENE *scene

The `Scene` implementation of `"free"` (cf. 5.4.4.1) sends `"delete"` to all instances which belong to this scene, i.e. to `object`, `next_scene` and `suprop`.

GRAPE(scene, "display")()

SCENE *scene

If an instance of class `Scene` receives a message `"display"` all transformations will be executed and the corresponding method will be send to the objects. The `"display"` affects the standard graphic device.

GRAPE(scene, "universal")(method)

SCENE *scene
char *method

The message method is send to the next_scene and to the object. Furthermore this method behaves like `"display"`.

(SCENE *)GRAPE(scene, "expand")()

SCENE *scene

Sometimes it is useful to combine a list of scenes into one new node or expand an existing geometry object to a scene. The new scene contains the old scene and all connected instances which then can be handled as one scene. The return value is the pointer to the new scene.

GRAPE(scene, "reflect-send")()

SCENE *scene

This method creates a new scene by an interactive procedure. The necessary object will be chosen from the scene's list of objects by picking (cf. method "find-pick") into the actual graphic device. At last this object will be reflected at a line (defined by two points) or a plane (defined by three points). The points are defined by picking, too. The new object will be integrated in the scene as the last node.

This picking is controlled with the mouse buttons:

```
left button   -- mark the points one after another
middle button -- abort
right button  -- perform reflection
```

To pick a certain point means to choose the nearest point of the geometry in the current scene with the actual projection on the graphic window.

GRAPE(scene, "reflect-delete-send")()

SCENE *scene

This method removes the last scene in the actual scene's list. If the actual scene was extended by reflections this scene is the last reflected object.

GRAPE(scene, "find-pick")(v, u, w, obj, dist)

INSTANCE *scene
VEC3 v
double u, w, *dist
INSTANCE **obj

In the instance scene a point v is searched so that this point is nearer to the straight (u, w, .) than dist. The global transformation matrices are taken into account. If a point is found then *obj is set to the scene the object with this point belongs to, v to this nearest point and dist the new distance. Combined with the method "pick" this one is useful to pick points or objects with the mouse. It has to be assured that the method "find-pick" is implemented on all scene's objects.

GRAPE(scene, "interpol")(scene2, f, mat1, mat2)

SCENE *scene
SCENE *scene2
double f
MATRIX44 mat1, mat2

According to the method `"display"` an instance of class `Scene` is displayed on the actual standard device. The object's data are transferred directly without creating any segments. The new object is an temporarily generated interpolation of both objects according to the following rule:

```
scene * mat1 * (1 - f) + scene2 * mat2 * f
```

The parameters mat1 and mat2 define the transformations performed on the corresponding objects before the interpolation takes place. If NULL is provided the identical transformation is performed.

(SCENE *)GRAPE(Scene, "read")()

> **SCENE *scene**
> **CLASS *Scene**

`"read"` reads instances of class `Scene` (written by the obsolete `"write"` method) from the standard textdevice.

7.1.1.2 Methods for `Scene` handling

Interactive methods sometimes need to handle objects in data hierarchy tree without knowing its structure. The following methods of class `Manager` can help you doing this.

GRAPE(mgr, "goto-instance")(object)

> **MANAGER *mgr**
> **INSTANCE *object**

Search `object` in the data hierarchy tree and take the minimal number of steps to go there.

(SCENE *)GRAPE(mgr, "insert-object")(object, bury, at_head, skip_first)

> **MANAGER *mgr**
> **INSTANCE *object**
> **int bury, at_head, skip_first**

Insert `object` into the data hierarchy tree at a good place, as `object` of a `Scene` instance. If the scene found already has an object a new `Scene` instance is created and inserted. `object` is `ASSIGN`ed to the `Scene`, which is returned upon successful completion.

The flags decide what is thought to be a good place:

bury if set, follow the `next_scene` links of scenes to insert the object as deep as possible. If a `Scene` instance without object is found on this way put `object` there.

at_head always insert the object at the top of the data hierarchy tree. You need this or the following flag if you call `"insert-object"` from a `"-send"` method that will be called on the new object by the universal mechanism.

skip_first take the second scene found on the way to the data hierarchy tree top.

Some old style methods which insert new objects into the data hierarchy tree are implemented as service methods ob class `Scene`, that only work on objects of special classes. Now you can implement these methods on the correct class and use `"insert-object"` to insert the results.

This method uses

(SCENE *)GRAPE(scene, "insert-object")(object, bury)

> **SCENE *scene**
> **INSTANCE *object**
> **int bury, at_head, skip_first**

Insert `object` as `object` of a `Scene` instance. `bury` is the same as in the method of class `Manager`.

7.1.1.3 Suprop

The class Suprop stores the information dealing with the rendering style and the surface properties. Its main purpose is to give certain scene nodes surface properties.

(SUPROP *)GRAPE(suprop, "read")()

(SUPROP *)GRAPE(Suprop, "read")()

> **SUPROP *suprop**
> **CLASS *Suprop**

The method `"read"` builds up a new instance containing the information read from the standard text device (if send to the class) or fills an existing one (if send to an instance). It only exists for backward compatibility reasons, use XDR files instead (cf. 5.7).

(SUPROP *)GRAPE(suprop, "store")()

(SUPROP *)GRAPE(Suprop, "store")()

GRAPE(suprop, "load")()

GRAPE(scene, "store-suprop")()

> **SUPROP *suprop**
> **CLASS *Suprop**
> **SCENE *scene**

The method `"store"` stores the current surface properties of the standard graphic device in the instance scene. If `"store"` is send to the class a new instance of this class is returned containing the appropriate information. The method `"load"` does it vice versa. The method `"store-suprop"` stores the current surface properties in the instance variable scene->suprop.

7.1.2 Chain

A subclass of `Scene` is `Chain`. `Scene` is mainly used as a binary node whereas nodes of class `Chain` are used to store a linear sequence of objects as they arise for example when generating multiple copies of an object during reflection operations. Methods on the class `Chain` assume to find such a linear structure.

```
typedef struct chain {
    ...
    /* superclass structure */
    int flag;           /* only used for reflection at the moment:
                              MARKED:     reflect chain
                              NOT-MARKED: don't reflect */
} CHAIN;
```

7.1.2.1 Memory Methods on Chain

GRAPE(chain, "hardcopy")()

> **CHAIN *chain**

Make an identical copy of chain: a new chain (with a hardcopied object) is created by sending "hardcopy" to the superclass `Scene` and copying the flag. The new `Chain` instance is returned.

GRAPE(chain, "softcopy")(new_chain)

> **CHAIN *chain, *new_chain**

Make a softcopy of chain: like "hardcopy" but the object is softcopied (only the used data is copied). If new_chain is not NULL then this instance is used to store the copy otherwise a new `Chain` instance is created.

GRAPE(chain, "merge-send")()

CHAIN *chain

Merge chain with all its next-scene links, i.e. merge chain->object with chain->next-scene ->object into chain->object , then change the pointer next-scene of chain to chain->next-scene ->next-scene and send "delete" to the old chain->next-scene and so on.

GRAPE(chain, "merge")(chain2)

CHAIN *chain
CHAIN *chain2

Merge chain and chain2, i.e. merge chain->object with chain2->object to chain->object and continue with chain->next-scene and chain2->next-scene and so on.

7.1.2.2 Other Methods on Chain

GRAPE(chain, "get-object")(chain_inter, time)

CHAIN *chain
CHAIN *chain_inter
double time

This method assumes that chain has components on which also a "get-object" method exist. Then a copy of chain is returned in chain-inter but with the result of sending "get-object" to chain ->object as objects instances. If chain-inter is NULL a new chain is allocated and returned.

GRAPE(chain, "get-link")(obj_index)

CHAIN *chain
int obj_index

This method returns the obj_index_th link in chain. If no chain is found NULL is returned.

(INSTANCE *)GRAPE(chain, "get-link-step")(obj-index, time)

CHAIN *chain
int obj-index
double time

This method assumes that the instance object points to an instance of class TimeStep. It searches the obj_index-th link in the chain and searches in (TIMESTEP *)link->object for the step with the next smaller step->time than time.

(INSTANCE *)GRAPE(chain, "get-link-pre-obj")(obj_index, time)

(INSTANCE *)GRAPE(chain, "get-link-post-obj")(obj_index, time)

CHAIN *chain
int obj_index
double time

Like `"get-link-step"` but instead of returning the timestep its pre or post object is returned.

GRAPE(chain, "universal")(method)

CHAIN *chain
char *method

The method method is send to chain->object and all of its next-scenes objects.

GRAPE(chain, "interpol")(chain1, chain2, factor)

GRAPE(chain, "interpol-fac")(chain1, chain2, fac1, fac2)

CHAIN *chain
CHAIN *chain1, *chain2
double factor
double fac1, fac2

This methods assumes that chain1 and chain2 have objects on which also "interpol" methods exist. Then an "interpolated" chain is returned ((1 - factor) * chain1 + factor * chain2 respectively fac1 * chain1 + fac2 * chain2) which has the result of sending "interpol" to chain1->object and chain2->object as chain->object.

GRAPE(chain, "refine")(time)

GRAPE(chain, "coarsen")(time)

CHAIN *chain
double time

Refine or coarsen the objects of chain and all of its next-scenes around time by sending "refine" or "coarsen" to chain->object and chain->next-scene.

(CHAIN *)GRAPE(chain, "cons-close")()

CHAIN *chain

During "close-send" on a `Geom2d` instance some parameter lines or boundary curves may become unused. These lines are detected and removed with this routine. Be careful: on return the original list pointer may have changed when the first list link is removed, therefore the new pointer is returned.

GRAPE(chain, "find-pick-index")(v, obj_index, index, u, w, obj, dist)

CHAIN *chain
VEC3 v
int *obj_index, *index
double u, w double *dist
INSTANCE **obj

Search for a point `v` in the chain `chain` for which the distance from the straight line `(u, w, .)` to `mat` * `v` is less than `sqrt(dist)`. If such a point is found *`obj` is set to the found chain, `v` to the new point, *`index` to the index of the point, *`obj_index` to the index of the picked object and *`dist` to the new distance. The picked chain instance is returned.

7.1.3 TimeScene

A further subclass of `Scene` is `TimeScene`. A node in the hierarchy of class `TimeScene` has additionally a local time and a pointer to a dynamic object. The object of the `TimeScene` is always a section at the current local time of the dynamic process. This is the simplest way to include dynamic objects into the hierarchy.

```
typedef struct timescene {
    ...
    /* superclass structure */
    double time,object_time;
```

```
    INSTANCE *dynamic;
    char *dynamic_method_name;
    int sync;
} TIMESCENE;
```

7.1.3.1 Memory Methods on TimeScene

GRAPE(timescene, "hardcopy")()

TIMESCENE *timescene

Make an identical copy of timescene: a new timescene (with hardcopied dynamic and object) with the same time and object_time is created by sending "hardcopy" to the superclass `Scene` and to timescene->dynamic.

GRAPE(timescene, "softcopy")(new-tsc)

TIMESCENE *timescene, *new-tsc

Make a softcopy of timescene: like "hardcopy" but object and dynamic are softcopied (only the used data is copied). If new_tsc is not NULL then this instance is used to store the copy otherwise a new TimeScene instance is created.

7.1.3.2 Other Methods on TimeScene

GRAPE(timescene, "universal")(method)

TIMESCENE *timescene
char *method

Send parameter free methods (send– and disp–methods) to timescene->dynamic which is updated first by calling "get-object".

GRAPE(timescene, "display")()

TIMESCENE *timescene

Like "universal" but the method `"display"` is send to timescene->next_scene, too.

GRAPE(timescene, "switch-sync-send")()

TIMESCENE *timescene

Toggle the timescene->sync flag (1 means synchronized, 0 not synchronized).

GRAPE(timescene, "synchronize")(time)

TIMESCENE *timescene
double time

Set timescene's and all of its `next_scene`'s time to time.

GRAPE(timescene, "follow")()

TIMESCENE *timescene
INSTANCE *follow[3][3]
char pop-push[3][3]

Set the environment data of the instance timescene for the object hierarchy in the interactive manager.

GRAPE(timescene, "get-info")(xlist)

TIMESCENE *timescene
XLIST *xlist

Create info list entries for a TimeScene instance. Entries are class, name, refcount, next scene, object type and name, matrix type, display method, surface properties, dynamic name, time and sync status.

GRAPE(timescene, "goto-time-send")()

TIMESCENE *timescene

Send `"goto-time"(timescene->time)` to the dynamic object, hoping this method will place you on the current key frame using `"goto-instance"` of `Manager` (7.1.1.2).

This method is designed for interactive use.

(TIMESCENE *)GRAPE(timescene, "read")()

(TIMESCENE *)GRAPE(TimeScene, "read")()

TIMESCENE *timescene
CLASS *TimeScene

These methods read a `TimeScene` instance to (from) the standard text device and return *NULL* in case of failure.

If "read" is sent to the class, the data is read into a new instance, otherwise into the receiver *timescene*. In each case the method returns the new timescene (pointer).

These methods only exist for backward compatibility reasons, use XDR files instead (cf. 5.7).

7.2 Moving and Tracing

7.2.1 The Concept of Moving and Tracing

In the introductional section we have described the application of static visualization techniques on time cuts of dynamical processes. But these visualization techniques on time cuts are not the only possible approaches to get a deeper insight into the behaviour of interesting phenomena in time. A different approach examines integrated evaluations on the process. They are also of great importance in numerical approaches based on the method of characteristics. Let us think for example of a flow problem, where a vector field v describes the movement of particles or sets of the latter in time. In detail this means that a particle x at time t_0 in a domain g moves according to the first order differential equation $\partial_t x(t) = v(x(t), t)$ with the initial condition $x(t_0) = x$. The stationary visualization methods allow us to inspect this field at certain times from the point of view of a fixed observer, where we can interactively vary time and method parameters in the way it is described in other sections. In comparison to that the integration techniques let us have a closer look on the evolution itself. This means we follow sets of particles in time and record there behaviour in some sense in terms of a moving observer. In the framework of continuum mechanics these two different visualization approaches reflected the two main different coordinate systems, the Lagrangian and the Eulerian coordinates. Now using the integration technique we have two possibilities at hand. The first and standard one is to record only the trace of a set of particles and throw away the time information. The second gives us a complete overview on the movement of the specific set of particles in time. Therefore we extract the movement again as a dynamical process. This new dynamical process then contains the positions and the corresponding times of one particles or of a set of particles, a curve or a surface. We can look at this as a subset of the information of the initial process in which we have performed the integration. Although these two approaches look different from the rendering point of view they can be condensed in one calculation operation.

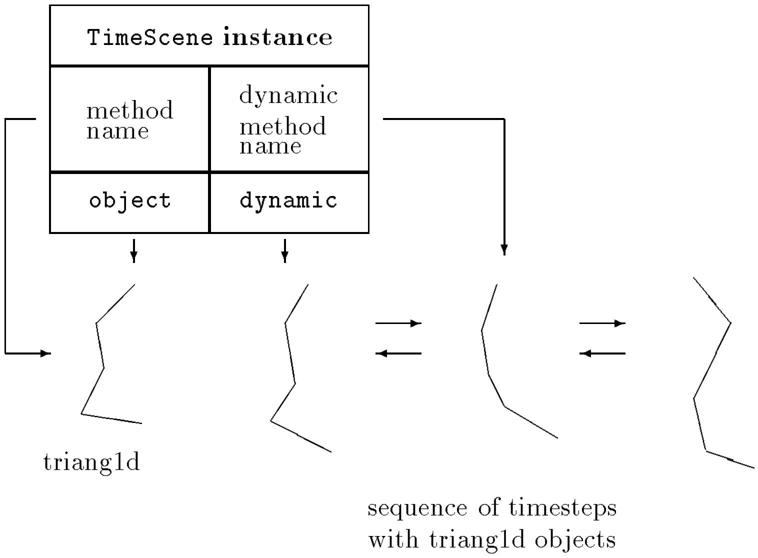


Figure 7.1: example of tracing

Let us call it *moving*. *Moving* always produces a time dependent process *move* containing the evolution (the movement) of the specific set in space we are interested in. This process is again an instance of class `TimeScene`. Visualizing this as a trace or as time cuts of the *move* is then a question of the `"display"` method on the new process and not of the calculation. In details this means we always calculate moves and then we render them in two different ways. Up to now we only know display methods working on `Scene` or `TimeScene` objects, this means on time cuts of the evolution process *move*. Regarding the possibility to define a `"display"` method named **`dynamic_method_name`** working on the instance variable `dynamic`, as it is mentioned in the previous section in the description on `TimeScene`, allows us to display the trace of the process *move*. We will call this kind of rendering *tracing*.

To be more explicit let us briefly examine the underlying equation. If $S \subset \Omega$ is the set to be moved in the field v defined on Ω we want to calculate a one parameter family of sets $S(t)$ which is defined as follows

$$\begin{aligned} S(t) &= \{x(t)|x(t_0) \in S\} \quad \text{where} \\ \partial_t x(t) &= v(x(t), t) \quad , \qquad x(t_0) = x \end{aligned}$$

If $S(t)$ consists of only one point we discretize the differential equation in time. What is actually used is a variant of the second order Euler Cauchy scheme. With the general well known Euler Cauchy scheme we mean the following second order one step ODE method. Its local consistency is of order $O(\delta_t^3)$.

$$\begin{aligned} v_1 &= v(x(t), t) \\ x_1 &= x(t) + \delta_t \; v_1 \\ v_2 &= v(x_1, t + \delta_t) \\ x(t + \delta_t) &= x(t) + \delta_t \; \frac{1}{2}(v_1 + v_2) \end{aligned}$$

We will use the δ_t for the small timestep width in the ODE scheme. Further on we will not store all these timestep but only a few of them. This is possible because of the interpolation techniques working on this list of steps. We will call this larger step width Δt but underlining that it does not depend on the timestep of the process in which we move. The main disadvantage of this standard ODE scheme working on mesh data is that it ignores the special structure of the mesh. The scheme above for example has a consistency error of $O(\delta_t^3)$ if the data is smooth. But the vectorfield which is in our case the data is in general not smooth. For viscous flow problems attacked by conforming Finite Element strategies it is only C^0 at the edges or boundary surfaces of the elements and for highly inviscous flow numerics and non–conforming Finite Elements it is even worse. We find smoothness only in the interior of the elements. This shows that the standard ODE solves are useless for general situations on discrete meshes. A typical example for the failure of this straightforward approach is the calculation of particle traces passing a shock with a shift in the velocity where according to an adaptation strategy the shock area is intensively refined. The modified scheme we use overcomes this problem. It has a local consistency order of

$$O(\min(\delta_t\ ,\ h)^3)$$

where h is a local upper bound of the discretizations in space. Fore details we refer to a forthcoming paper.
Now we should turn to the general case were $S(t)$ is no longer a single point. We discretize in space the set S by a triangulated approximation S^{δ_h} were δ_h is a bound for the size of the simplices. Each vertex point of this set will then be moved according to the ODE scheme mentioned above. This defines sets $S^{\delta_h}(i\,\Delta t)$ which should approximate $S(i\,\Delta t)$. The initial approximation error of S^{δ_h} with respect to a smooth set S is of the order $O(\delta_h^2)$. But without any mesh adaptation on the meshes S^{δ_h} we can in general not suppose $S^{\delta_h}(t)$ to be a good approximation of $S(t)$. Therefore we apply an adaptive strategy to refine the discretizations at each timestep following the lines of the adaptive algorithm described in section about dynamical processes and starting with $S^{\delta_h}_{post}(t) = S^{\delta_h}$.

$$\begin{aligned} S^{\delta_h}_{pre}(t+\Delta t) &= \mathrm{OneStep}(v, S^{\delta_h}_{post}(t)) \\ S^{\delta_h}_{post}(t+\Delta t) &= \mathrm{AdaptMesh}(S^{\delta_h}_{pre}(t+\Delta t) \end{aligned}$$

Here **OneStep** is one iteration of the described ODE scheme on the vertex points based on the modified **EulerCauchy** scheme.

$$x(t+\delta_t) \;=\; \mathrm{EulerCauchy}(v,t,\delta_t,x(t)) \quad \forall_{x(t)}x(t) \text{ vertex in } S^{\delta_h}_{post}(t)$$

Here the **AdaptMesh** has to keep aware of degenerating angles as well as of enlarging elements. Let us finally resume that the time scales of the original process and of an arbitrary extracted move process are totally independent.

7.2.2 Calculating a Move

Now let us first discuss the calculation techniques for the *move* processes. In detail a *moving* method works on a `TimeScene` or `Scene` node containing the stationary or nonstationary process in which we want to move a set of points. It creates or completes a `TimeScene` node containing the *move*. If created this `TimeScene` is inserted as `next_scene` in the `scene` or `TimeScene` of the process in which we want to move. There are three different moving methods at hand. The `"point-moving"` produces a `TimeScene` containing the movement of the point in an instance of an appropriate class `Trace`. The starting point is fixed by picking. The *first-moving* method takes the `object` of the `next_scene` node of the current process. This `object` might be a curve or

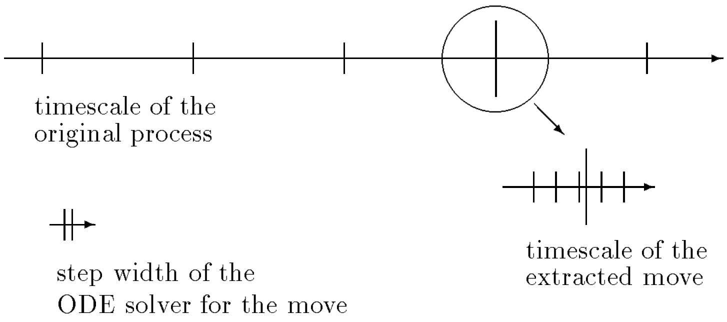


Figure 7.2: the different scales

a surface. This object forms the set to be moved. A `TimeScene` containing a list of timesteps as its `dynamic` instance variable is created. The timesteps in there have curves or surfaces as `pre_object` and `post_object`. They are the shapes of the moved starting set at a time `timestep ->time`. The third method is called *continue-moving*. It adds the recalculated time cut of the move at a specific time. This method allows to refine the calculated *move* by inserting a timestep in between the existing sequence. Having in mind interactive flexibility and computing efficiency we differ between point and curve or surface *moving*. The methods and their parameters will be described later.

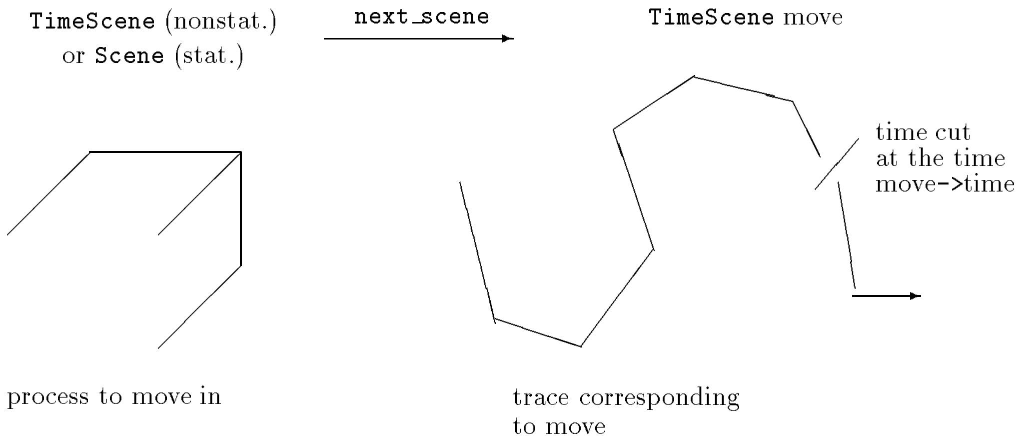


Figure 7.3: moving process

Let us now turn to a detailed description of the basic methods calculating the move.

Let us call the initial time t_0 for a moving calculation from now on `start_time`. Corresponding to that the calculations stops at the `end_time`. The project (section 8.1.1) *move-trace* contains a number of buttons and rulers steering the calculation and its parameters. Therefore the user has to add this project first. The main parameters the user can adjust are the following

- `number_of_steps`
  the number of steps to be calculating in `"first-moving"` or `"point-moving"`
- `start_time`
  where the moving starts in `"first-moving"` or `"point-moving"`
- `end_time`
  where the moving ends in any moving method

- `refine`
  a maximum percentage of elements of the moved geometry the moving is allowed to refine per step

The methods behind the included buttons work on scenes and TimeScenes as well. The `object` of a scene is regarded as the stationary data set to move in and the `dynamic` of a TimeScene as the nonstationary one. A general parameter `runge_delta` fixes the step length of the integration algorithm.

The Moving Methods

GRAPE(s, "first-moving")()

SCENE *s

This method moves a given test set (a surface or a curve) located at the object position of the `next_scene` and possibly positioned by a model transformation `s->object_trans`. It produces `number_of_steps` timesteps. The object to be moved could be read from disk or in the case of a Triang1d object selected by the picking method `"mark-polygon-send"`. This method is implemented on scenes with an underlying `Triang2d` or `Triang3d` object. The move starts at time `start_time`, produces equi distant timesteps and ends at time `end_time`. The created sequence of timesteps forms the `dynamic` instance variable of a `TimeScene` instance inserted as `s->next_scene`. It is an adaptive data set and the discretization changes between two timesteps if a specific adaptation criteria is fulfilled.

GRAPE(s, "continue-moving")()

SCENE *s

This method inserts a new timestep in the precalculated list of timesteps located at `s->next_scene ->dynamic`. Usually this list has been generated by an application of the method `"first-moving"`. The `"continue-moving"` searches the timestep `step` with the largest time smaller then `end_time`. Then `step->post_object` is moved and the resulting object is inserted as a timestep at time `end_time`. Here `step->post_object` and `step->next_step->pre_object` have the same underlying discretization. Therefore no adaptation is possible while inserting timesteps in between the moving sequence.

GRAPE(s, "point-moving")()

SCENE *s

The method `"point-moving"` deals with the moving of single particles. It allows the flexible picking of points. For 3D objects the picking is performed on the intersection plane of the planar clipping methods. This single point is then moved in the same manner as other higher dimensional sets by the `"first-moving"` method. The resulting `s->next_scene->dynamic` is an instance of the class `Trace` containing the moving information. The user can preview particle moves and if they seem interesting enough it is finally possible to insert them in the hierarchy. The class of the created `dynamic` instance variable on the next node is `Trace` because a list of timesteps with points as `pre_object` or `post_object` is not handy and efficient enough. `Trace` is a subclass of `Triang1d`.

GRAPE(tr, "refine-moving")(refine)

TRIANG1(2)D *tr

This method refines the geometry to be moved according to some appropriate criteria such as the smoothness and regularity of the discretization. This operation takes place after each move in `"first-moving"`. It must be ignored in `"continue-moving"`. Otherwise the interpolation between the inserted timesteps and its `post_step` would be undefined. The argument `refine` gives a maximum percentage for the elements of the moved geometry which should be refined.

Triang0d

The class `Triang0d` allows to handle single points as objects. It is not design for an efficient handling of large sets of points. This would be the task for another class. `Triang0d` fills a gap in the set of classes for simplex geometries in different dimensions. It is for example used as the time cut generated by the method `"get-object"` on an instance of the class `Trace`.

```
#CLASS Triang0d : Root {
  double x,y,z;
};
```

GRAPE(t0, "display")()

TRIANG0D *t0

In the grid and patch mode a small sphere makes the single point visible. The radius of the sphere is included as a ruler in the option menu of the manager.

(TRIANG2D *)GRAPE(tr2d, "xdr")(xdrp)

CLASS *Triang0d;
TRIANG0D *tr0d;
XDR *xdrp;

For the I/O of `Triang0d` instances XDR is used. To get more information about XDR the eXtended Data Representaion see 5.7.

Trace

For efficiency reasons the timedependent behaviour of a point is not collected in a huge list of timesteps with points as `pre_object` or `post_object` but in one data structure containing the position of the points $(x, y, z)(t)$ and the corresponding times t in arrays. We name the corresponding class `Trace`. The structure `Trace` for an instance of this class look as follows

```
#CLASS Trace : Triang1d {
  double *t;
};
```

GRAPE(trace, "display")()

TRACE *trace

In the grid mode of the graphic device the curve is drawn as a line. In the patch mode this line is rendered as a pipe with a leading arrow at the top and a completing half ball at the tail. The parameters are included in the option menu of the manager as rulers. For further details we refer to the documentation of the same method on the class Triang1d.

(TRACE *)GRAPE(trace, "read")()

(TRACE *)GRAPE(Trace, "read")()

(TRACE *)GRAPE(trace, "xdr")(xdrp)

TRACE *trace
CLASS *Trace
XDR *xdrp;

A `Trace` instance is read from the standard textdevice. If `"read"` fails it returns `NULL`. This method only exists for backward compatibility to read files written with `"write"` and is replaced by the **GRAPE**–xdr device. To get more information about XDR the eXtended Data Representaion see 5.7.

GRAPE(trace, "shorting")(eps)

TRACE *trace
double eps

This method modifies the points in a `Trace` instance by throwing away points which distance to their neighbours are smaller then `eps`. The resulting `Trace` instance has therefore a minimum distance between two neighbour points of `eps` size.

7.2.3 Visualizing the Trace

In the previous sections the *moving* methods have been described as the fundamental calculation methods extracting the movement of sets of particles in a stationary or nonstationary process. They create a dynamical process containing the movement. The `object` of this `TimeScene` is the time cut of the moved geometry. Different display methods may be invoked on this object by changing the `display_method` in the `TimeScene`.

But it is also possible to render the total `Trace` of the moved geometry in various ways. This display method can be stored in the instance variable `dynamic_method_name` of the `TimeScene` containing the movement. There are the following *tracing* methods at hand.

The Tracing Methods

GRAPE(trace, "tracing")(start_time,end_time)

GRAPE(trace, "tracing-disp")()

TRACE *trace
double start_time,end_time

The trace is rendered only from `start_time` to `end_time`. The rendering style is the same as for the method `"display"`. In the interactive case `start_time` and `end_time` are identical to the `start_time` and `end_time` of the *moving* method.

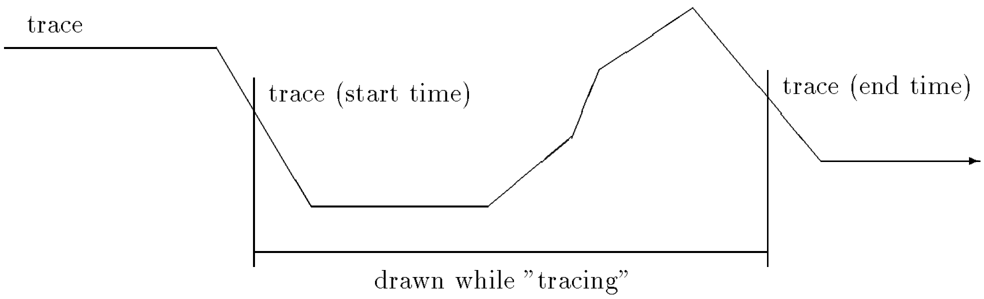


Figure 7.4: tracing

GRAPE(timestep, "tracing")(start_time,end_time)

GRAPE(timestep, "tracing-disp")()

TIMESTEP *timestep
double start_time,end_time

This method first checks whether the objects in the sequence of timesteps are of a subclass of Triang1d or not. If not the method is ignored. Otherwise a segment of the trace of a moving curve is rendered from `start_time` to `end_time`. The rendering style is that of a triangulated surface. In the interactive case `start_time` and `end_time` are identical to the `start_time` and `end_time` of the *moving* methods.

7.2.4 Clouds

The following class Cloud a subclass of Root is dedicated for the animation of larger sets of particles. A timestep list of Cloud objects stores this information. The typedef structure for a cloud instance is as follows:

```
#CLASS Cloud : Root {
  int number_of_points;
  VEC3 *x;
};
```

GRAPE(cloud, "display")()

CLOUD *cloud

In the grid mode for each point of the cloud a cross with lines in x,y and z direction is drawn. In patch mode every point is rendered as a small ball. The radius can be scaled in the interactive environment varying a ruler.

GRAPE(cloud, "interpol")(cloud1,cloud2,lambda)

CLOUD *cloud,cloud1,cloud2
double lambda

An interpolated Cloud object represents the convex combination in between two given cloud objects `cloud1`, `cloud2`.

CLOUD *GRAPE(tr2d,"2cloud")()

CLOUD *GRAPE(tr1d,"2cloud")()

CLOUD *GRAPE(s,"2cloud-send")()

SCENE *s
CLOUD *cloud
Triang1d *tr1d
Triang2d *tr2d

Besides explicit generation of cloud objects by specific methods there are conversion routines at hand to convert curves and surfaces of `Triang1D` or `Triang2D` type to `Cloud` objects containing solely the edges of the latter geometries. The `"2cloud-send"` method on scene invokes the sending of `"2cloud"` message to the scene object and replaces the latter by the generated cloud.

7.2.5 Streaklines

Streaklines describe the motion of an injected mass less material at a specific point in a flow volume over a certain time interval. Therefore a `Streakline` is a family of line segments. The following class `Streakline` holds data of a single Streakline:

```
#CLASS StreakLine : Trace {
  int number_of_steps;
  int max_number_of_steps;
};
```

where `number_of_points` is the number of points in a single trace segments (the trace for a single particle, released at one instant in time). There are `number_of_steps` time intervals under consideration. This results in `number_of_steps` +1 single traces. A time cut of the streakline is a line build up by the corresponding cuts of the single traces. With increasing time its length grows first and then keeps at a certain length. A list of times gives the stored timecuts of a streakline. `number_of_steps` is assumed to be greater than `number_of_points`.

```
------------ time ---------->

-----------------|-->
 -single trace ---|--->
  ----------------|---->
   ---------------|----->
    --------------|------>
     -------------|------->
      ------------|-------->
       -----------|--------->
        ----------|---------->
                  streakline
```

A `streak time` ruler in the moving projects allows to fix the length of the single traces, where the interval from `start time` to `end time` determines the total considered time interval. A button `"move streaks"` allows to release streaklines at picked positions.

GRAPE(streak, "display")()

STREAKLINE *streak

The method `"display"` visualizes the first trace in a streakline.

GRAPE(streak, "get-object")(triang1d,t)

STREAKLINE *streak
TRIANG1D *triang1d
double t

`"get-object"` returns a Triang1d instance containing the current time slice at time `t`.

GRAPE(streak, "list-alloc")(maxpoints,maxsteps)

GRAPE(streak, "list-free")()

STREAKLINE *streak
int maxpoints,maxsteps

`"list-alloc"` allocates memory for a maximum of `maxpoints` points in a single trace of the `Streakline` and `maxsteps` integrations intervals. This results in a maximum of `maxsteps` +1 such traces. `"list-free"` frees the memory in a `Streakline`. For more information about `"list-alloc"` and `"list-free"` see 5.4.4.2.

GRAPE(streak, "get-times")(stime,etime)

STREAKLINE *streak
double *stime,*etime

`get-times` fills `stime`, `etime` with the maximal and minimal time of a `Streakline`.

7.3 TimeStep

The class `TimeStep` is currently the only fully implemented dynamic process in **GRAPE**. In this section we will only document the methods on `TimeStep`, refer to section 4.4.4 for a description of how to use this class.

`TimeSteps` are nodes in a double linked list (the dynamic process), they have a pointer to the previous (`pre_step`) and the following (`post_step`) step. The data stored in each `TimeStep` are the `time` of the step and the two objects `pre_object` and `post_object`:

```
#CLASS TimeStep : Root {
  double time;
  struct timestep *pre_step, *post_step;
  INSTANCE *pre_object, *post_object;
};
```

The `pre_object` and `post_object` of a `TimeStep` can either be identical (the simple case) or point to different instances. But even in the second case both objects have to represent the same data, for triang objects for example the geometries (FE data etc.) must be identical, only the discretizations may differ, see figure 4.22.

The most important method on `TimeStep` is

(INSTANCE *) GRAPE(step, "get-object")(inst, time)

> **TIMESTEP *step**
> **INSTANCE *inst**
> **double time**

It creates a current object for the given `time` from the `TimeStep` sequence `step`. This object is created by interpolation therefore an `"interpol"` method must exist on the class of the `step`'s objects. The result of the interpolation is stored in the `inst` instance and returned, if `inst` is `NULL` an new instance is allocated instead.

Since `"get-object"` uses `"interpol"` to create the object for a specific time from the dynamic process it must be possible to interpolate the `post_object` of a `TimeStep` and the `pre_object` of the following step `post_step`, for the triang classes therefore the discretizations of these objects have to be identical.

7.3.1 Memory Methods on TimeStep

GRAPE(step, "free")()
GRAPE(step, "zero-refs")()

> **TIMESTEP *step**

`"free"` deletes all `pre_objects` and `post_objects` of the `TimeStep` sequence `step` (i.e. all objects of the `pre_steps` and `post_steps`, too), finally the `TimeSteps` itself are freed.

Since the steps in the sequence point to each other (`step->post_step`'s `pre_step` is `step`) the `TimeStep` class needs a special `"zero-refs"` method (compare 5.4.5.2) to determine if the sequence can be deleted.

GRAPE(step, "hardcopy")()
GRAPE(step, "softcopy")(copy)

> **TIMESTEP *step, *copy**

Both methods create a copy of `step` with the same number of steps and the same times as `step` and return it. `"softcopy"` uses a given `copy` sequence to store the copy, if `copy` is `NULL` it allocates new `TimeSteps` for the copy as `"hardcopy"` does.

The big difference between these methods is that `"hardcopy"` copies all objects of the `step` sequence using `"hardcopy"` while `"softcopy"` only `ASSIGNs` the objects of the `step` sequence to the `copy` sequence, existing objects in the `copy` sequence are deleted.

7.3.2 Methods for Creating `TimeStep` Sequences

(TIMESTEP *) GRAPE(TimeStep/step, "put")(pre, post, t)

> **CLASS *TimeStep**
> **TIMESTEP *step**
> **INSTANCE *pre, *post**
> **double t**

This method adds a new `TimeStep` with `pre` and `post` as its `pre_object` and `post_object` and time `t` to an existing `TimeStep` sequence `step` or — if called on the class `TimeStep` — creates a new `TimeStep`. If there already is a step with time `t` in the `step` sequence only this existing step is returned (no new step is created), otherwise `"put"` returns the new step. Note that `pre` and `post` are `ASSIGN`ed to the appropriate variables so if these objects go out of scope afterwards you have to send them `"delete"` to get the number of references right.

An example:

```
{
  TIMESCENE *tsc;
  INSTANCE *pre, *post, *first, *last;
  double t = 0.0;

  tsc = (TIMESCENE *)GRAPE(TimeScene, "new-instance")(<<name>>);

  first = <<read/create object>>(t);
  tsc->dynamic = (INSTANCE *)GRAPE(TimeStep, "put")(first, first, t);
  GRAPE (first, "delete")(); /* now first is only referred by the TimeStep */
  while (t < 1.0) {
    t += delta;
    pre = <<pre-object for time>>(t);
    post = <<post-object for time>>(t);
    tsc->dynamic = (INSTANCE *)GRAPE (tsc->dynamic, "put")(pre, post, t);
    GRAPE (pre, "delete")();
    GRAPE (post, "delete")();
  }
  /* t >= 1.0 */
  last = <<last object>>(t);
  GRAPE (tsc->dynamic, "put")(last, last, t);
  GRAPE (last, "delete")();
}
```

Of course one can delete `first` or `post` a little later if they are needed to construct the next objects:

```
{
  TIMESTEP *tsp;
  MYCLASS *obj, *result, *next;
  double t = 0.0;
  obj = <<the object>>;
  tsp = (TIMESTEP *)GRAPE(TimeStep, "put")(obj, obj, t);
  while (t < 1.0) {
    result = (MYCLASS *)GRAPE (obj, "change-time")(t);
    GRAPE (obj, "delete")();
    t += delta;
    next = (MYCLASS *)GRAPE (result, "change-geometry")(t);
    GRAPE (tsp, "put")(result, next, t);
    GRAPE (result, "delete")();
    obj = next;
  }
  ...
```

}

(TIMESTEP *) GRAPE(TimeStep, "alloc")(num, tstart, tend, name)

> **CLASS *TimeStep**
> **int num**
> **double tstart, tend**
> **char *name**

In contrast to `"put"` which just adds one step to an existing sequence `"alloc"` creates a complete `TimeStep` sequence with `num` steps with equidistant times in the interval `[tstart, tend]`. The first step gets the name `name`, all following steps are named `step i/num` with `i` in `[2, num]`.

Only the `TimeSteps` itself are created, to create objects for the sequence you can use

GRAPE(step, "create-objects")(class, name)

> **TIMESTEP *step**
> **CLASS *class**
> **char *name**

This method creates for each `TimeStep` in the sequence `step` an object of class `class` with name `name` (the `pre_object` is created and then `ASSIGN`ed to the `post_object`).

GRAPE(step, "insert-step")(time)

> **TIMESTEP *step**
> **double time**

Insert a new `TimeStep` at time `time` in the sequence `step` and return it, if already a step for `time` exists nothing happens and `NULL` is returned.

If the `time` is within the time range of the `step` sequence `"insert-step"` creates the object for the new step by calling `"get-object"`(`NULL`, `time`), otherwise the first or last object of the sequence is copied. If the new step is inserted at the start or end of the sequence the times are renormalized.

7.3.3 Other Methods on `TimeStep`

GRAPE(step, "display")()
GRAPE(step, "universal")(method)

> **TIMESTEP *step**
> **char *method**

`"display"` sends `"display"` to the `pre_object` and — if it differs from the `pre_object` — to the `post_object` of `step`.

If `method` is a display method (i.e. has the suffix `"-disp"`) `"universal"` only sends it to the objects of `step`, other methods are send to the objects of all `TimeSteps` in the `step` sequence. As for `"display"` the `method` is only sent once if `pre_object` and `post_object` are identical.

GRAPE(step, "get")(time)
GRAPE(step, "get-step")(time)
(INSTANCE *) GRAPE(step, "get-pre-obj")(time)
(INSTANCE *) GRAPE(step, "get-post-obj")(time)
(INSTANCE *) GRAPE(step, "get-next-obj")(inst)

> **TIMESTEP *step**
> **double time**
> **INSTANCE *inst**

`"get"` returns the `TimeStep` with time `time` in the sequence `step` if it exists, otherwise it returns `NULL`. Be careful, this method isn't able to handle cyclic `TimeSteps` (first pointing to last and vice versa).

"get-step" returns the TimeStep in the step sequence with time less than or equal to time, if time is smaller than all times in the sequence the first step is returned. "get-pre-obj" returns the pre_object of the step with time greater than or equal to time, "get-post-obj" returns the post_object of the step with time less than or equal to time (the post_object of the step returned by "get-step").

To access all objects in the TimeStep sequence step "get-next-obj" can be used. If called with a NULL instance inst it returns the first object in the sequence (step->pre_object), otherwise it returns the next object following inst (the post_object of a step if inst is its pre_object and the pre_object and post_object differ else the pre_object of the following step). Be careful, NULL objects in the TimeStep sequence will get you into trouble.

GRAPE(step, "get-times")(stime, etime)

> **TIMESTEP *step**
> **double *stime, *etime**

This method determines the times of the first and the last TimeStep in the step sequence and stores them in stime and etime.

GRAPE(step, "goto-time")(time)

> **TIMESTEP *step**
> **double time**

Find the TimeStep belonging to time time and call "goto-instance" on the standard Manager to set it as the current instance (see 7.1.1.2). This method is for example called by "goto-time-send" on class TimeScene if step is set as dynamic object.

(TIMESTEP *) GRAPE(TimeStep/step, "read")()

> **CLASS *TimeStep**
> **TIMESTEP *step**

Read a TimeStep sequence from the standard textdevice and return the first step (NULL in case of an error). This method is only provided for backward compatibility to read files written with "write", it is replaced by

GRAPE(step, "xdr")(xdrp)

> **TIMESTEP *step**
> **XDR *xdrp**

Read the instance step from or write it to the XDR stream xdrp. This method should not be called directly, refer to 5.7 (expecially 5.7.3) for details.

Since the **GRAPE** XDR system handles references automatically all TimeSteps in the sequence are read or written recursively, see the method code in section 5.7.6.2.

GRAPE(step, "merge")(step_two)

> **TIMESTEP *step, *step_two**

Merge the pre_objects and post_objects of all TimeSteps in the sequences step and step_two into those of step. Of course the number of steps, their times and the geometry structures of the objects of the sequences have to be identical to be able to merge them.

GRAPE(step, "refine")(time)

> **TIMESTEP *step**
> **double time**

Refine the TimeStep sequence step at the given time. If time is between steps the post_object of the step before time and the pre_object of the following step have the same discretization, both are refined in the same way. If time is exactly on a step this step and its pre_step and post_step have to be refined. If necessary "refine" splits the pre_object and the post_object of the steps that are refined.

GRAPE(step, "equiangulate")(time, method)

> **TIMESTEP *step**
> **double time**
> **char *method**

As "refine" (see above) this method handles the connectivity in the TimeStep sequence step and sends the method (usually "equiangulate-send", see 6.3.1.4) to the appropriate objects. Currently time must be the time of one of the steps, if time is between steps only an error message is printed.

GRAPE(step, "get-info")(xlist)
GRAPE(step, "follow")(follow, pop_push)

> **TIMESTEP *step**
> **XLIST *xlist**
> **INSTANCE *follow[3][3]**
> **char pop_push[3][3]**

Refer to section 5.4.7.3 for an explanation of the "get-info" and "follow" methods.

"follow" adds the pre_step (bottom), pre_object (bottom right), post_object (top right) and post_step (top) instances to the step's environment in the manager.

"get-info" adds information about the times of the step, its pre_step and post_step and about the classes of the pre_object and post_object to the info xlist.

GRAPE(step, "get-bnd-box")(xmin, xmax, ymin, ymax, zmin, zmax)

> **TIMESTEP *step**
> **double xmin, xmax, ymin, ymax, zmin, zmax**

Compute the bounding box of the step's objects.

7.4 Dynamic Objects Supervising TimeStep

The following classes simplify the use of class TimeStep for various applications, they provide a large set of methods and a user-interface which allows to work with (most of) them without programming.

The classes correspond to the similar named projects described in sections 8.2.1– 8.2.6, writing input files and using the interface is also explained in these sections.

7.4.1 Time_Object

Time_Object is the simplest class besides TimeScene supervising TimeStep. Since the class of the TimeStep's geometry is arbitrary the user has to write his own "compute-send" routine to compute or read in the data.

Time_Object provides the basic functionality needed to navigate through the TimeSteps, to read description files and to handle int and time-dependent double parameter. For more information about Time_Object, its user-interface, editing variables and input files see the documentation of the *Time-Object* project in section 8.2.1.

7.4.1.1 The Time_Object Structure

Time_Object provides instance variables for the task, frame and time handling, a Time_Parm instance for time-dependent parameter, the geometry which consists of a list of TimeStep sequences and the geometry inter for the current time section.

```
#CLASS Time_Object : Root {
  TASK_TYPE task, task_mask; /* Task description, needed
                                for computation etc          */
  int frame, num_of_frames;  /* Current frame number, total
                                number of frames             */
  int *status;               /* Pointer to memory for frame
                                status (COMPUTED, MERGED etc)*/
  double *time, *time_step;  /* Pointer to current time and
                                memory for the time of each frame  */
  TIME_PARM *time_parm;      /* Time_Object parameter (see below)   */
  CHAIN *geometry, *inter;   /* TimeSteps with Time_Object geometry */
                             /* Interpol instance for displays etc */
  SCENE *tree;
};
```

The `task` must be `tskGeometry` for `Time_Objects`, for subclasses which provide more than one geometry more tasks are available (`tskDomain` corresponds to the `domain` geometry of `Surface`, `tskNewton` correspond to the `newton` geometry of `Amandus` and so on):

```
typedef enum {
  tskGeometry,  /* bitwise flags */
  tskGauss,
  tskConjugate,
  tskDomain,
  tskNewton,
  tskExplicit,
  tskAny
} TASK_TYPE;
```

To create or update the `inter` instance which holds the time section of the `geometry` for the current `*time` the method `"get-object"` is used, see section 4.4.4.

7.4.1.2 Time_Parm

The `Time_Object` parameter are stored in a `Time_Parm` instance:

```
#CLASS Time_Parm : Root {
  /* PRIVATE */
  BOUNDED_RULER **int_rul;  /* Ruler for integer parameter,
                                                  not time dependent */
  FUNCTION_RULER **dbl_rul; /* Function ruler for double parameter */
  /* USER */
  int num_int_parm;         /* Number of integer parameter */
  int num_double_parm;      /* Number of double parameter  */
  int *int_parm;            /* Pointer to memory for integer parameter */
  double *double_parm;      /* Pointer to memory for double parameter  */
};
```

Each `int`/`double` parameter corresponds to a `Ruler`/`Function_Ruler` which allows to edit its value. The `double` variables are time-dependent, the `Splines` of the associated `Function_Rulers` describe their values as functions over the time, see 4.1.2.6.

There are only a few methods on `Time_Parm` for memory management, debugging, for (de)activating the rulers and for archiving the `int` and `double` values:

GRAPE(tparm, "free")()

TIME_PARM *tparm

Delete an instance of class `Time_Parm`. All memory is freed using `"list-free"`, the rulers are removed from the manager and deleted. See 5.4.4.1 for details on `"delete"`/`"free"`.

GRAPE(tparm, "list-alloc")()
GRAPE(tparm, "list-free")()

TIME_PARM *tparm

Allocate or free the memory for the `int` and `double` variable arrays and the corresponding `Ruler` arrays. `num_int_parm` and `num_double_parm` have to be set before `"list-alloc"` is called.

The rulers itself are not created by `"list-alloc"`, this is usually done by the function `g_time_parm_get_data` (see below) which is called when a description file is read. `"list-free"` removes the rulers and deletes them before freeing the arrays.

GRAPE(tparm, "get-info")(xlist)

TIME_PARM *tparm
XLIST *xlist

Add entries for the numbers of parameter and the parameter values to the superclass information, see section 5.4.7.3.

GRAPE(tparm, "set")(state)
GRAPE(tparm, "switch-send")()
GRAPE(timeobj, "switch-comp-send")()

TIME_PARM *tparm
int state
TIME_OBJECT *timeobj

`"set"` activates (`state == ON`) or deactivates (`state == OFF`) all `tparm` rulers, `"switch-send"` toggles their state. `"switch-comp-send"` is the `Time_Object` counterpart of `"switch-send"`, it just calls this method on `timeobj->time_parm`.

GRAPE(tparm, "xdr")(xdrp)

TIME_PARM *tparm
XDR *xdrp

Read the `tparm` variables from or write them to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

The rulers are currently not handled by this method, therefore the `double` variables will lose their time-dependency.

There are some utility function which help reading and handling the `Time_Parm` variables:

```
int g_time_parm_get_data(FILE *fp, TIME_PARM *tparm,
                         char *name, char *path, double *time)
```

reads in the number of `int` and `double` parameter and their initial values (see 8.2.1.3 for an example file), creates the corresponding rulers and adds them to the manager.

```
int g_time_parm_get_var_list(TIME_PARM *tparm, char *varlist)
```

simplifies parsing formulas which use the `tparm` variables. After calling this function the string `varlist` contains a list of the variable names separated by spaces (see 8.2.2.1 for naming conventions), it starts with `"t"` for the time followed by the `int` parameter `"i ..."` and the `double` parameter `"a ..."`.

```
void g_time_parm_get_var_values(TIME_PARM *tparm, double time,
                                double *values)
```

helps evaluating functions. It stores the `tparm` variable values for the current `time` in the `values` array: `values[] = { time, (double)int_parm[0], ..., (double)int_parm[num_int_parm - 1], double_parm[0], ..., double_parm[num_double_parm - 1] }`.

7.4.1.3 Memory Methods on Time_Object

(TIME_OBJECT *) GRAPE(Time_Object, "new-instance")(name)

CLASS *Time_Object
char *name

Create an instance of class `Time_Object` with name `name`. `"new-instance"` only sets the `task` to `tskGeometry`, initializes the `time` with `"get-time"` (see 7.4.1.7) and creates the `time_parm` instance.

GRAPE(timeobj, "free")()

TIME_OBJECT *timeobj

Delete a `Time_Object` instance. All memory is freed by calling `"list-free"` and deleting `time_parm`. Refer to 5.4.4.1 for more information about `"delete"`/`"free"`.

GRAPE(timeobj, "list-alloc")()
GRAPE(timeobj, "list-free")()

TIME_OBJECT *timeobj

`"list-alloc"` allocates the memory for `status` and `time_step` for `num_of_frames` frames (this value has to be set before calling `"list-alloc"`!). The `status` for all frames is set to `NOT_ACTIVE`, the frame times are initialized with equidistant values in `[0, 1]`. `"list-alloc"` frees all memory but `time_parm` which was allocated in `"new-instance"` and is therefore deleted in `"free"`.

Of course `"list-alloc"` cannot allocate memory for the `geometry` since its class is arbitrary. The user has to provide a `"compute-send"` method (see 7.4.1.5) to allocate the memory and to create the data.

GRAPE(timeobj, "hardcopy")()
GRAPE(timeobj, "softcopy")(copy)

TIME_OBJECT *timeobj, *copy

Create a copy of `timeobj` and return it. The only difference between these methods is that `"softcopy"` uses the `copy` instance to store the copy, if `copy` is `NULL` a new instance is allocated.

Since `"hardcopy"` resp. `"softcopy"` is used to copy the `geometry` the result might differ, e.g. if the class of the underlying geometry is `Geom2d` (see 6.3.4.1).

`inter` and `tree` of the copy are set to `NULL`, `time_parm` is not copied but only `ASSIGN`ed (this results in some problems and might be changed in the future).

7.4.1.4 Display Methods on Time_Object

GRAPE(timeobj, "display")()
GRAPE(timeobj, "geometry-disp")()

TIME_OBJECT *timeobj

Display the time section of the `geometry` for time `*time`, it is created by calling `"get-object"` (see 4.4.4) and stored in `inter` which then is displayed.

Both methods are identical (in fact `"display"` just calls `"geometry-disp"`), but subclasses with more than one geometry might replace `"display"` and then the `geometry` still can be displayed with `"geometry-disp"`.

GRAPE(timeobj, "multi-disp")()
GRAPE(timeobj, "multi-display")()
GRAPE(timeobj, "do-multi-disp")(geom)

TIME_OBJECT *time_object
CHAIN *geom

These methods split the graphics window in several sub-windows and calls "display" for different times for each of them. "multi-disp" adds some rulers to the option menu which allow to control the time-range (relative to the current time), the number of pictures and an additional horizontal and vertical distance between them. "muli-disp" then calls "multi-display" which then calls "do-multi-disp" with geometry as parameter.

This construction may seem a litte bit complicated, but is simplifies implementing this method for subclasses: only "multi-display" which calls "do-multi-disp" with the correct parameter has to be provided.

7.4.1.5 Compute Methods on Time_Object

GRAPE(timeobj, "compute-video-send")()

TIME_OBJECT *timeobj

Compute all frames for timeobj by sending "compute-send" to each frame. The current frame number and time are saved and restored after the computation.

GRAPE(timeobj, "compute-send")()

TIME_OBJECT *time_object

This is the method which computes the Time_Object's data. Most of the memory management has to be done here because the class of the timeobj geometry is arbitrary. If the user wants to compute his own Time_Objects he can replace this method with his own "compute-send" method.

The default "compute-send" method which computes a torus-knots is presented in section 8.2.1.3 together with the corresponding input file.

GRAPE(ruler, "parameter-changed")()

RULER *ruler

This method is called when a Time_Object parameter ruler is changed (be careful, changing a double parameter also changes the associated spline). All frames of the Time_Object the ruler belongs to will be recomputed by calling "compute-video-send", this allows to change the Time_Object interactively.

7.4.1.6 I/O Methods on Time_Object

GRAPE(timeobj, "read-data")(path)
(char *) GRAPE(timeobj, "get-extension")()

TIME_OBJECT *timeobj
char *path

"read-data" reads in the Time_Object description file path/name.to where name is the name of the timeobj instance. g_time_parm_get_data (see section 7.4.1.2) is used to read the parameter data and to create the associated rulers, then "list-alloc" is called (by then "num_of_frames" is known) and all frames are computed.

"read-data" is usually called by "read-in" on class FileList, the default method of the read *.in button. The button pops up a file-requester which shows all files with the extension returned by "get-extension", for Time_Step this extension is *.to (its really a pattern and not an extension, but for historic reasons the method is called "get-extension").

All subclasses of Time_Object which use their own input files (with a different extension) have to implement these methods.

GRAPE(timeobj, "xdr")(xdrp)

TIME_OBJECT *timeobj
XDR *xdrp

Read the timeobj instance from or write is to the XDR stream xdrp. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

Only the time variable is not handled by this method. This should be no problem since the time-dependency of the double variables is lost anyway because of Time_Parm "xdr"....

7.4.1.7 Time_Object Task, Frame, Time and File Handling

Most of the methods described in this section are used by the project *Time-Object* user-interface, they provide the basic task, frame, time and file handling.

(GROUP *) GRAPE(Time_Object, "get-panel")()
(METHLAYER *) GRAPE(Time_Object, "get-display-layer")()
(TIME_OBJECT *) GRAPE(Time_Object, "update-panel")()

CLASS *Time_Object

These method are used by the *Time-Object* project to create and maintain the user-interface.

"get-panel" returns the interface shown in figure 8.13, "update-panel" is called when the current object of the manager is changed to find a Time_Object for the interface.

"get-display-layer" returns a layer which allows to select different display methods for Time_Objects, see figure 8.15.

Both the user-interface and the display layer are automatically added to resp. removed from the manager when the project *Time-Object* or one of its subprojects is added resp. removed.

(CYCLE_BUTTON *) GRAPE(Time_Object, "get-task-button")()
GRAPE(timeobj, "set-task")(task)
GRAPE(timeobj, "update-task")()

CLASS *Time_Object
TIME_OBJECT *timeobj
TASK_TYPE task

"set-task" sets the task for timeobj to task (of course this only makes sense for subclasses of Time_Object which allow more than one task). It then calls "update-task" which updates the task button which can be obtained by "get-task-button" (this method is for example used by "get-panel" to access the task button).

(GROUP *) GRAPE(Time_Object, "get-frame-buttons")()
GRAPE(timeobj, "set-frame")(frame)
GRAPE(timeobj, "update-frame")()
GRAPE(Time_Object, "previous-frame")()
GRAPE(input, "change-frame")()
GRAPE(Time_Object, "next-frame")()

CLASS *Time_Object
TIME_OBJECT *timeobj
INPUT *input
int frame

"set-frame" sets the frame number for timeobj to frame, then it calls "update-frame" to update the frame number on the input field in the group of frame buttons which can be obtained by "get-frame-buttons" (compare the task methods above).

The three methods "previous-frame", "change-frame" and "next-frame" are the default actions for the frame buttons, they allow to step through the frames or to select a specific frame by typing in its number.

(RULER *) GRAPE(Time_object, "get-time-ruler")()
GRAPE(timeobj, "set-time")(time)
GRAPE(timeobj, "update-time")()
(double *) GRAPE(Time_Object, "get-time")()
(double *) GRAPE(timeobj, "get-time-var")()

> **CLASS *Time_Object**
> **TIME_OBJECT *timeobj**
> **double time**

`"set-time"` sets the `*timeobj->time` to `time`, then it calls `"update-time"` which updates the time ruler for `Time_Objects` and all time-dependent variables of the `timeobj`. The time ruler can be optained by `"get-time-ruler"`, it is usually added to the option menu by the *Time-Object* project.

The method `"get-time"` is used to initialize the `time` instance variable of a `Time_Object` instance when it is created. It returns a pointer to the global `Time_Object` time variable (the one the ruler obtained by `"get-time-ruler"` is bound to). It is possible to change the `time` variable of a `Time_Object` instance — this should be done before the description file is read —, in this case only the local (scene) time ruler in the manag menu can be used. This ruler is bound to the `double` variable returned by `"get-time-var"`, for `Time_Objects` this method returns `timeobj ->time`.

(GROUP *) GRAPE(Time_Object, "get-stdin")()
GRAPE(filelist, "create-read-in")()
GRAPE(filelist, "read-in")()
GRAPE(timeobj, "write-in")()

> **CLASS *Time_Object**
> **FILELIST *filelist**
> **TIME_OBJECT *timeobj**

`"get-stdin"` returns the pointer to the group containing the `read *.in` and `write *.in` buttons, it is used by `"get-panel"` to add these buttons to the *Time-Object* user-interface.

`"read-in"` and `"write-in"` are the default methods for the buttons. `"read-in"` pops up a file-requester which allows to select a description file for reading. The requester is filled by calling `"create-read-in"` which in turn calls `"get-extension"` to determine which files should be displayed (all `*.to` files for class `Time_Object`). `"write-in"` is still not implemented, pressing the button will print an error message.

7.4.1.8 Other Methods on `Time_Object`

GRAPE(timeobj, "merge-send")()

> **TIME_OBJECT *timeobj**

Send `"merge-send"` to the `timeobj`'s `geometry` to create a single object from the objects in the `Chain` sequence and deactivate all `time_parm` rulers.

This method assumes that a `"merge"` method exists on the class of the `timeobj geometry ->object`, compare for example `TimeStep` (7.3.3) and `Geom2d` (6.3.4.5) `"merge"`.

GRAPE(timeobj, "close-send")()

> **TIME_OBJECT *timeobj**

Send `"close-send"` to the `timeobj geometry->object`, if necessary (if `geometry->next_scene` is not `NULL`) `"merge-send"` is called first. This method assumes that a `"close-send"` method exists on the class of the `geometry->object`.

See the description of the *Time-Object* user-interface in section 8.2.1.1 for further explanations.

GRAPE(timeobj, "universal")(method)

TIME_OBJECT *timeobj
char *method

`Time_Object` `"universal"` sends display methods to the `inter` instance (which is updated first by calling `"get-object"`) and all other methods to the `geometry`.

See section 5.4.8 for more information about `"universal"` methods.

GRAPE(timeobj, "get-info")(xlist)
GRAPE(timeobj, "follow")(follow, pop_push)

TIME_OBJECT *timeobj
XLIST *xlist
INSTANCE *follow[3][3]
char pop_push[3][3]

Refer to section 5.4.7.3 for an explanation of the `"get-info"` and `"follow"` methods.

`"follow"` adds the `time_parm` (top right) and `geometry` (right) instances to the `timeobj`'s environment in the manager.

`"get-info"` adds information about the frame, the number of frames, the task and the current time of the `timeobj` to the info `xlist`.

GRAPE(timeobj, "get-bnd-box")(xmin, xmax, ymin, ymax, zmin, zmax)

TIME_OBJECT *timeobj
double *xmin, *xmax, *ymin, *ymax, *zmin, *zmax

Compute the bounding box of the `timeobj` `inter` geometry using `Scene` `"get-bnd-box"`. `"get-bnd-box"` exists for all triang classes, if you use you own class for the `geometry` you have to provide a `"get-bnd-box"` method.

If you send `"insert-bnd-box"` to a `Scene` instance the bounding box for the `Scene`'s `object` is inserted as the `Scene`'s `next_scene`.

GRAPE(timeobj, "insert-step-send")()

TIME_OBJECT *timeobj

Insert a new frame for time `*time` into the `timeobj`'s `geometry`, the current `frame` is set to this new frame.

This method uses `"insert-step"` on `Chain` and `TimeStep`. The new frame is computed by interpolation, if the step is inserted at the end of the `geometry`'s `TimeStep` sequence the times of the other steps are rescaled so that all times are in `[0, 1]`.

GRAPE(timeobj, "mark-object-send")()
GRAPE(timeobj, "mark-geometry-send")()

TIME_OBJECT *timeobj

Since accessing the `timeobj`'s `inter` instance (the current time section which is for example used by `"display"`) is difficult, these two methods are provided: `"mark-object-send"` marks `inter` and `"mark-geometry-send"` marks `inter->object` — this has the same effect as pressing the `mark` button in the manag menu (see section 4.3.5.3).

GRAPE(timeobj, "reset-send")()

TIME_OBJECT *timeobj

Reset the status of the `timeobj` by sending `"reset-send"` to the `geometry` and by setting the `status` of all frames to `NOT_ACTIVE`.

This method is only needed for subclasses of `Time_Object` like `Amandus` which provide several geometries and allow to compute them step by step, calling `"reset-send"` then allows to compute already computed geometries again.

GRAPE(timeobj, "is-time-dependent")()

TIME_OBJECT *timeobj

This method just returns the `timeobj` (the `"is-time-dependent"` method on `Root` returns `NULL`) since `Time_Object`'s are time-dependent.

This method is used to find a time-dependent object when the current object of the manager is changed, the ruler for the local (scene) time is then bound to this object.

GRAPE(timeobj, "syncronize")(time)

TIME_OBJECT *timeobj
double time

Same as `"set-time"` (see next section), used by the manager to update all time-dependent instances when the local (scene) time ruler is changed.

7.4.2 Time_Curve

Class `Time_Curve` is the simplest subclass of `Time_Object`. It is designed for working with time-dependent curves, in contrast to `Time_Object` it uses a specific data structure: the `geometry Chain` has a `TimeStep` sequence as `object`, the objects of the `TimeSteps` are of class `Triang1d`. There are only a few methods on `Time_Curve`, most functionality — even the user-interface — is inherited from the superclass.

The structures and methods for `Time_Curve` are defined in the *Curve* project described in section 8.2.2, therefore the `typedefs` for `TIME_CURVE` and `CURVE_PARM` are not part of the `grape.h` header file (you shouldn't need them anyway). All sources for this project can be found in demo directory.

7.4.2.1 The Time_Curve Structure

`Time_Curve` only adds three pointers to coordinate functions and information about the time-dependent definition interval and the distribution of the points on the curve to its superclass.

```
typedef struct {
  TIME_OBJECT_STRUCT;                  /* Subclass of Time_Object */
  CURVE_PARM *curve_parm;              /* Curve parameter (see below) */
  struct { t_fun *x, *y, *z; } coord_f; /* Coordinate functions */
  } coord_f;
} TIME_CURVE;
```

The data for `curve_parm` and `coord_f` are usually read from a description file with extension `*.cv`, the functions are read as strings, compiled into a `t_fun` with the function compiler and evaluated during the computation (refer to section 10.2.8 for a description of the function compiler). The `time_parm` variables can be used for the functions, see section 8.2.2.1.

7.4.2.2 Curve_Parm

The time-dependent definition interval and the distribution of points are stored in a `Curve_Parm` instance:

```
typedef struct {
  INSTANCE_STRUCT;              /* Subclass of Root */
  FUNCTION_RULER *min_rul;      /* Internal ... */
  FUNCTION_RULER *max_rul;
  FUNCTION_RULER *dstrbn_rul;
  t_fun *t_min, *t_max;         /* ... use only min, max and dstrbn */
```

```
  int num_of_points;            /* Number of points */
  SPLINE1D *dstrbn;             /* Spline for distribution of points */
  double min, max;              /* Definition interval */
} CURVE_PARM;
```

You should only use `min`, `max` and the `dstrbn Spline`. The rulers are used to edit these values, they are added to the option menu when a description file is read in. The two `t_fun`s are only used to set up the ruler splines for `min` and `max`, to describe these values in the input file a time-dependent function (i.e. a function depending on `t`) can be used.

There are only some memory management functions on class `Curve_Parm`:

GRAPE(cparm, "free")()

CURVE_PARM *cparm

Delete an instance of class `Curve_Parm`. All data is freed using `"list-free"`, all `cparm` rulers are removed from the manager and deleted. See 5.4.4.1 for details on `"delete"`/`"free"`.

GRAPE(cparm, "list-alloc")()
GRAPE(cparm, "list-free")()

CURVE_PARM *cparm

`"list-alloc"` is only provided for consistency, all memory for `Curve_Parm` instances is (usually) done in `"read-data"` which reads a `Curve_Parm` input file.

`"list-free"` frees all data including the function rulers and the spline which have been allocated in `"read-data"`.

7.4.2.3 Memory Methods on `Time_Curve`

(TIME_CURVE *) GRAPE(Time_Curve, "new-instance")(name)

CLASS *Time_Curve
char *name

Create an instance of class `Time_Curve` with name `name` by calling `"new-instance"` on the superclass and allocating the `Curve_Parm` instance `curve_parm`.

GRAPE(tcurve, "free")()

TIME_CURVE *tcurve

Delete a `Time_Curve` instance. All memory is freed by calling `"list-free"`, deleting `curve_parm` and calling `"free"` on the superclass. For details on `"delete"`/`"free"` refer to 5.4.4.1.

GRAPE(tcurve, "list-alloc")()
GRAPE(tcurve, "list-free")()

TIME_CURVE *tcurve

`"list-alloc"` allocates the superclass data by calling `"list-alloc"` on `Time_Object` and it creates the `geometry` for `tcurve` (`num_of_frames` and `time_curve->num_of_points` have to be initialized before `"list-alloc"` is called). `num_of_frames` frames with `Triang1d` objects are allocated for `geometry` using `TimeStep` `"alloc"` (see 7.3.2) and `"create-objects"` (see 7.3.2), additionally `curve_parm->num_of_points` points are allocated for each `Triang1d` instance. `"list-free"` frees all the memory by sending `"list-free"` to `curve_parm` and the superclass.

7.4.2.4 Other Methods on `Time_Curve`

Besides of `"get-info"` the methods described in this section are exactly what is needed for a subclass of `Time_Object`: a method `"get-extension"` that determines which files belong to the class, `"read-data"` to read in description files (both methods are called by the `"read-in"` method bound to the `read *.in` button) and a `"compute-send"` method which computes the data:

GRAPE(tcurve, "read-data")(path)
(char *) GRAPE(tcurve, "get-extension")()

> **TIME_CURVE *tcurve**
> **char *path**

`"read-data"` reads in the `Time_Curve` description file `path/name.cv` — all `Time_Curve` description files should have the extension `*.cv` returned by `"get-extension"` — where `name` is the name of the `tcurve` instance. Additional to the same method on `Time_Object` (compare 7.4.1.6) the number of points for the curve, the definition interval and the coordinate functions are read, see section 8.2.2 for an example description file.

GRAPE(tcurve, "compute-send")()

> **TIME_CURVE *tcurve**

Compute the points of `tcurve` using the definition interval, the distribution of points from `curve_parm` and the coordinate functions from `coord_f`.

GRAPE(tcurve, "get-info")(xlist)

> **TIME_CURVE *tcurve**
> **XLIST *xlist**

Add information about the number of points of the `Triang1d` instances to the superclass information.

7.4.3 Surface

This class is the fundamental subclass of `Time_Object` for time-dependent surfaces. `Time_Object` uses almost the same structure for the geometries as `Time_Curve` (see 7.4.2), the only difference is that the objects of the `TimeSteps` in the `Chain` sequences are of class `Geom2d`.

Class `Surface` should be used as an "abstract class": it provides much functionality for subclasses, but there are no description files for `Surface` itself and no `"compute-send"` method to compute any data.

7.4.3.1 The Surface Structure

In contrast to `Time_Object` and `Time_Curve` which only uses one `geometry`, class `Surface` provides additional geometries:

```
#CLASS Surface : Time_Object {
  CHAIN *model;               /* This is the model which is used for
                                 almost all operations (computation,
                                 reflection etc). Depending on the
                                 current TASK, model points to
                                 geometry, domain, gauss or conjugate */
  CHAIN *domain;              /* Chain of timesteps for domain */
  CHAIN *gauss;               /* Chain of timesteps for gauss */
  CHAIN *conjugate;           /* map and conjugate surface */
  CHAIN *conj_inter;          /* Interpol instance for display etc */
  double *asso;               /* Pointer to current associate value */
  REFLECTION_CONTROL *reflect;   /* Used for reflection */
};
```

These geometries are used to store the `domain`, the `gauss` map, i.e. the stereographic projection of the normals onto the plane, and a `conjugate` geometry which usually has the same connectivity as the `geometry`.

For `Surface` and its subclasses the `task` is used to select one of the geometries as the current `model` (compare section 7.4.1.1), changing the task with `"set-task"` will also change the `model`. Most methods which work on a single geometry use the `model`, `"display"` for example computes the current time section from the `model` and stores it in `inter`.

Like `inter` for the `model`, the `conj_inter` instance is usually used to store the current time section of the `conjugate` geometry. Together with the `asso` variable which can be used to define an interpolation between `inter` and `conj_inter` this allows to visualize a two-parameter family of surfaces, see 7.4.3.4 for an example.

7.4.3.2 Reflection_Control

Class `Surface` provides methods to reflect the time-dependent geometries on (time-dependent) planar or straight symmetry lines, points, normals or circles. The `Reflection_Control` instance `reflect` handles the reflection `type` and selection of `objects` for the reflection:

```
#CLASS Reflection_Control : Root {
  int type;         /* bitwise flag for reflection type:
                          REFL_ALL (no object selection if set),
                          REFL_AUTOMATIC (type guessed if set),
                          REFL_PLANAR, REFL_LINE, REFL_POINT,
                          REFL_NORMAL, REFL_CIRCLE (fixed type) */
  int obj;          /* bitwise flag for objects to be reflected:
                          REFL_MODEL, REFL_GEOMETRY, REFL_DOMAIN,
                          REFL_GAUSS, REFL_CONJUGATE, REFL_NEWTON */
  int last_obj_index;
  int object_index, bdy_index, bdy2_index;
};
```

The *Surface* project (see section 8.2.3) provides an interface which allows to select `type` and `objects`, and to select boundary curves of the `Geom2d` instances (which are used to store the data of the geometries) for the reflections.

To be able to reflect a time-dependent geometry only the indices of the selected boundary curve(s) are used, they are stored in the `..._index` variables.

There are only two methods on `Reflection_Control`:

GRAPE(rflctrl, "init")()
GRAPE(surface, "reflect-init")()

> **REFLECTION_CONTROL *rflctrl**
> **SURFACE *surface**

`"init"` is used to initialize the `type` and `obj` with the values selected in the user-interface. `"reflect-init"` just calls `"init"` on `surface->reflect`, it is called on the current `Surface` when the reflect configuration layer shown in figure 8.17 is closed.

GRAPE(rflctrl, "xdr")(xdrp)

> **REFLECTION_CONTROL *rflctrl**
> **XDR *xdrp**

Read the `rflctrl` variables from or write them to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (expecially 5.7.3) for details.

7.4.3.3 Memory Methods on Surface

(SURFACE *) GRAPE(Surface, "new-instance")(name)

> **CLASS *Surface**
> **char *name**

Create an instance of class `Surface` with name `name`. Additional to the superclass method the `asso` variable is initialized by calling `"get-asso"`, to allocate memory for the geometries use `"list-alloc"`.

GRAPE(surface, "free")()

SURFACE *surface

Free the memory of a `Surface` instance by sending `"list-free"` to it and `"free"` to the superclass. See 5.4.4.1 for details on `"delete"`/`"free"`.

GRAPE(surface, "list-alloc")(points, bnds)
GRAPE(surface, "list-free")()

SURFACE *surface
int points, bnds

`"list-alloc"` initializes all `Surface` instance variables by calling the superclass `"list-alloc"` method — therefore `num_of_frames` has to be set before `"list-alloc"` is called — and creating `reflect` and the `Chains` for the geometries and `conj_inter`.

For the `objects` of the geometries `geometry`, `domain`, `gauss` and `conjugate` sequences of `num_of_frames` frames with `Geom2d` instances as objects are created using `TimeStep` `"alloc"` and `"create-objects"`. For each `Geom2d` instance memory for `points` points, `2*points` elements and `bnds` boundary curves is allocated.

`"list-free"` frees all memory of the `surface` instance by deleting all geometry instance variables and calling `"list-free"` on the superclass `Time_Object`.

GRAPE(surface, "hardcopy")()
GRAPE(surface, "softcopy")(copy)

SURFACE *surface, *copy

Create a copy of `surface` and return it. From the geometrical point of view there is only one real difference between these two methods: `"softcopy"` uses a given `copy` to store the copy, if `copy` is `NULL` it allocates a new instance (as `"hardcopy"` does). From the physical point of view the copies created by `"hardcopy"` and `"softcopy"` differ since they use `"hardcopy"` resp. `"softcopy"` to copy the superclass data and the geometries.

7.4.3.4 Display Methods on `Surface`

GRAPE(surface, "display")()

SURFACE *surface

Display the time section of the `model` for time `*time`. The time section it is created by calling `"get-object"` on `model`, is is stored in `inter` and then displayed.

GRAPE(surface, "domain-disp")()
GRAPE(surface, "gauss-disp")()
GRAPE(surface, "conjugate-disp")()

SURFACE *surface

These methods allow to display the different geometries of `surface` independent from the current `model`. As `"display"` these methods store the time section in `inter` and display it.

GRAPE(surface, "associate-disp")()
GRAPE(surface, "associate")(asso)
(double *) GRAPE(Surface, "get-asso")()

SURFACE *surface
CLASS *Surface
double asso

Display the associate surface for the `surface` `geometry` and `conjugate`. `"associate-disp"` adds a ruler which controls the global associate variable to the option menu and then calls `"associate"` which with the value of `*surface->asso`. `"associate"` first creates the time sections of `geometry` and `conjugate` and stores them in `inter` and `conj_inter`. The associate surface is computed by interpolation of `inter` and `conj_inter`

```
cos(asso) * inter + sin(asso) * conj_inter
```

and then displayed. `"new-instance"` initializes the `surface->asso` variable of `Surface` instances with the global associate variable returned by `"get-asso"`, if you change `surface->asso` you have to add you own ruler to be able to edit the `asso` value interactively.

GRAPE(surface, "multi-display")()

SURFACE *surface

This method just calls `"do-multi-disp"` with the `model` as argument, see 7.4.1.4.

GRAPE(surface, "symmetry-disp")()

SURFACE *surface

This display method expects `surface` to consist of parts of a triply periodic surface bounded by planar symmetry lines. `"symmetry-disp"` first displays the `model` by calling `"display"` — this also updates the `inter` instance —, then it uses `"compute-symmetries"` on `Geom2d` (see 6.3.4.5) to compute the symmetry polyhedra for all `objects` in the `inter` `Chain` sequence and displays them.

GRAPE(surface, "explode-disp")()

SURFACE *surface

Never seen a surface explode? Just experiment with the rulers inserted into the option menu and observe the strange dependency on the time ruler.

7.4.3.5 I/O Methods on `Surface`

GRAPE(surface, "read-data")(path)
GRAPE(surface, "get-extension")()

SURFACE *surface
char *path

These methods are provided even though there are no description files for class `Surface`y, the extension for `Surface` files is `*.sf` (refer to section 7.4.1.6 for these methods on `Time_Object`).

GRAPE(surface, "xdr")(xdrp)

SURFACE *surface
XDR *xdrp

Read the `surface` instance from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

This methods handles all `Surface` instance variables but `asso`. If you use an `asso` variable which is different from the one returned by `"get-asso"` you have to change it after reading in data.

7.4.3.6 `Surface` Task, Reflection and Geom Status Handling

These methods are related to the project *Surface* user-interface described in section 8.2.3.

(GROUP *) GRAPE(Surface, "get-panel")()
GRAPE(Surface, "update-panel")()

CLASS *Surface

These methods are used by the *Surface* project to create and maintain the user-interface. `"get-panel"` returns the interface shown in figure 8.16, `"update-panel"` is called when the current object of the manager is changed to find a new `Surface` for the interface.

GRAPE(surface, "set-task")(task)
GRAPE(surface, "update")()

> **SURFACE *surface**
> **TASK_TYPE task**

`"set-task"` sets the `surface->task` to `task`. This will also change the `model` to the geometry corresponding to the task, e.g. `domain` for `tskDomain`. After changing the `model` `"set-task"` updates the `Geom2d` status button with `"update-geom-status"` (see below).

`"update"` calls `"set-frame"` and `"set-task"` to update time, frame number and task for an instance of class `Surface`. This is for example necessary if the `surface frame` or `task` was changed "by hand", e.g. after reading in a `Surface` instance.

(GROUP *) GRAPE(Surface, "get-reflect-buttons")()
GRAPE(surface, "reflect-send")()
GRAPE(surface, "reflect")()
GRAPE(surface, "reflect-delete-send")()

> **CLASS *Surface**
> **SURFACE *surface**

`"get-reflect-buttons"` returns a `Group` containing the three reflection buttons for the `Surface` interface, it is only used by `"get-panel"`.

`"reflect-send"` allows to interactively pick boundaries for the reflection and the objects to reflect, `"reflect"` then does the reflection. It reflects all or only the selected objects in the `model Chain`, the created objects are added at the end of the `model Chain`. If you make a mistake you can delete reflected objects with `"reflect-delete-send"`, the first time this method is called all objects created by the last reflection are deleted, on all subsequent calls only one object (the last in the `model Chain`) is deleted.

`"reflect-send"`/`"reflect-delete-send"` is called when the `reflect`/`refl: del last` button is pressed, for more information see 8.2.3.2.

(CHECKFIELD *) GRAPE(Surface, "get-geom-status-buttons")()
GRAPE(Surface, "new-geom-status")()
GRAPE(surface, "update-geom-status")()

> **CLASS *Surface**
> **SURFACE *surface**

`"get-geom-status-buttons"` returns a `CheckField` (see 9.1.3.3) which shows the status of the display flags of the current `Surface`'s `model Geom2d` instances (see 6.3.4), is is only used by `"get-panel"`.

`"new-geom-status"` is called if one of the `Checkbox`es is changed, it sets the display flags to reflect the new values. Conversely, `"update-geom-status"` has to be called when the display flags are changed, this will update the `Checkbox`es. For more information see 8.2.3.

7.4.3.7 `Surface` Methods for Creating Triangulations

These methods rely on equally or similarly named methods on `Chain`, `TimeStep` and `Geom2d` and their superclasses, they call them for the current frame or time or on all frames of the `model`. The project *Refine* provides an interface which allows to access most of these methods interactively, see section 8.2.8.

GRAPE(surface, "refine-send")()
GRAPE(surface, "coarsen-send")()

> **SURFACE *surface**

Call "refine-send"/"coarsen" send on the model geometry for the current time *time. Methods used are Chain "refine"/"coarsen"(which just pass on the method to their object and next_scene), TimeStep "refine"/"coarsen" (7.3.3) and the refine/coarsen methods on Adapt2d described in section 6.3.3.4.

GRAPE(surface, "refine-all-send")()

SURFACE *surface

Refine all frames in the model geometry using the same refine–field which is taken from the inter object and copied to all steps.

Using this method can be dangerous since all steps must have the same connectivity. If a method like "refine-send" or "equiangulate-send" which changes the connectivity locally (for some of the frames) was called before "refine-all-send" will result in an error, the model might be unusable. Another danger is that Adapt2d "refine-local-send" might use correction steps resulting in neighbour frames with different connectivities, in this case interpolation between these frames is no longer possible.

GRAPE(surface, "mollify-send")()
GRAPE(surface, "mollify")(mollify_method)

SURFACE *surface
char *mollify_method

"mollify" on Triangd or Geom2d improves the quality of the triangulation by replacing all points by the average of the point itself and its neighbours.

These two methods provide an interface to apply "mollify" to the surface model geometry: "mollify-send" calls "mollify" with "mollify-send" as argument, "mollify" then applies the method given as argument to the model. If "mollify-send" is used as a display method it allows to change several parameter, e.g. the number of times the mollify_method should be called and whether the mollify operation should be applied to boundary points, too.

Using two methods just to call one other might seem a bit strange but it allows to implement other mollify operation for subclasses of Surface, in this case only the "mollify-send" method which calls "mollify" with a different method as argument has to be implemented.

GRAPE(surface, "equiangulate-send")()

SURFACE *surface

This method just calls "equiangulate" on the model for the current *time. "equiangulate" on Chain just passes on the arguments, TimeStep "equiangulate" handles the connectivity (e.g. splitting pre_object and post_object if necessary). The methods which are finally called are "equiangulate-send" on Geom2d (6.3.4.5) and Triang2d (6.3.1.4).

7.4.3.8 Other Methods on Surface

GRAPE(surface, "compute-bndlen-send")()
GRAPE(surface, "compute-periods-send")()

SURFACE *surface

Compute the lengths of the boundary curves resp. the periods (if any) for all model frames and inter using the same methods on Geom2d (see 6.3.4.5).

GRAPE(surface, "merge-send")()
GRAPE(surface, "close-send")()
GRAPE(surface, "insert-step-send")()
GRAPE(surface, "reset-send")()

SURFACE *surface

Additional to the superclass methods described in section 7.4.1.8 these methods also handle the geometries `domain`, `gauss` and `conjugate`. `"reset-send"` also deletes reflected instances.

GRAPE(surface, "universal")(method)

> **SURFACE *surface**
> **char *method**

As the superclass method (see 7.4.1.8) display methods are send to the `inter` instance but other methods are send to the `model` instead of the `geometry`.

GRAPE(surface, "get-info")(xlist)
GRAPE(surface, "follow")(follow, pop_push)

> **SURFACE *surface**
> **XLIST *xlist**
> **INSTANCE *follow[3][3]**
> **char pop_push[3][3]**

Refer to section 5.4.7.3 for and explanation of the `"get-info"` and `"follow"` methods.

`"get-info"` adds information about the `*asso` value and the number of points and elements of the `Geom2d` instances of the `Surface` geometries to the superclass data.

`"follow"` adds the `domain` (top right), `model` (right) and `conjugate` (bottom right) geometries to the `surface`'s environment in the manager.

GRAPE(surface, "find-pick")(v, u, w, obj, dist)

> **SURFACE *surface**
> **VEC3 v**
> **double u, w**
> **INSTANCE **obj**
> **double *dist**

`"find-pick"` allows to pick a point in the current `inter Chain` of objects (if it doesn't exist `*obj` is set to `NULL`), the object and point found are returned in `obj` and `v`. This method uses `"find-pick"` on `Scene` (7.1.1.1) and `Triang2d` (6.3.1.4).

GRAPE(surface, "debug")()

> **SURFACE *surface**

Interactive debugging for `Surface` instances: points or objects can be picked with the mouse to get information about their coordinates, index, and transformation.

(CHAIN *) GRAPE(surface, "get-bnd-curve")(chain)

> **SURFACE *surface**
> **CHAIN *chain**

Extract the time-dependent boundaries from the `surface`'s `model Geom2d` instances and copy them to `chain`. `chain` is returned, if it is `NULL` a new `Chain` instance is created.

7.4.4 Surface_Build

The project *Build* described in section 8.2.4 is based on the `Surface_Build` subclass of `Surface`. `Surface_Build` adds an instance variable `build_parm` of class `Build_Parm` used for storing the description of surface patches to its superclass `Surface`.

`Surface_Build` is only available when the *Build* project is added — either interactively via the `add project` button in the manag menu or by calling `g_project_add` or `g_project_use`.

The type `SURFACE_BUILD` is not part of the **GRAPE** header file `grape.h`, but it isn't needed anyway. Methods should be implemented for the superclass `Surface`, if you really have to write a method for `Surface_Build` you can use `SURFACE` instead.

7.4.4.1 Memory Methods on Surface_Build

(SURFACE_BUILD *) GRAPE(Surface_Build, "new-instance")(name)

CLASS *Surface_Build
char *name

Create an instance of class `Surface_Build` with name `name` by calling `"new-instance"` on the superclass `Surface` and creating the `Build_Parm` instance `build_parm`.

GRAPE(sfbuild, "free")()

SURFACE_BUILD *sfbuild

Free the memory of a `Surface_Build` instance by sending `"delete"` to its `build_parm` instance and `"free"` to the superclass. For more information about `"delete"`/`"free"` see 5.4.4.1.

GRAPE(sfbuild, "list-alloc")()
GRAPE(sfbuild, "list-free")()

SURFACE_BUILD *sfbuild

`"list-alloc"` calculates the number of points needed for the geometries from the `build_parm` data (`build_parm` therefore has to be initialized before `"list-alloc"` is called) and then calls `"list-alloc"` on the superclass. If more than one surface patch is needed additional objects are added to the `geometry Chain`.

`"list-free"` just calls `"list-free"` on the superclass and on `build_parm`.

7.4.4.2 Compute Methods on Surface_Build

GRAPE(sfbuild, "compute-send")()

SURFACE_BUILD *sfbuild

Compute the surface patches based on the `build_parm` data. Depending on the kind of patch (triangle, quadrilateral), the triangulation type (standard or symmetric) and the boundary types different `"build-"` methods on `Geom2d` are used to compute the data.

7.4.4.3 I/O Methods on Surface_Build

GRAPE(sfbuild, "read-data")(path)
GRAPE(sfbuild, "get-extension")()

SURFACE_BUILD *sfbuild
char *path

`"read-data"` reads in the `Surface_Build` description file `path/name.bd` — all `Surface_Build` description files should have the extension `*.bd` returned by `"get-extension"` — where `name` is the name of the `sfbuild` instance. Depending on the point and patch data read from the file some rulers might be created and added to the option menu of the manager. The default method for the rulers is `"parameter-changed"` (see 7.4.1.5) therefore changing a ruler will recompute `sfbuild` for the new data.

GRAPE(sfbuild, "xdr")(xdrp)

SURFACE_BUILD *sfbuild
XDR *xdrp

Read the `sfbuild` instance from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

7.4.4.4 Other Methods on Surface-Build

GRAPE(sfbuild, "merge-send")()

SURFACE_BUILD *sfbuild

Additionally to merging all geometries by calling the superclass method (7.4.3.8) all `sfbuild` rulers are deactivated — recomputing a `Surface_Build` instance after merging it is not possible.

GRAPE(sfbuild, "assign-cons-send")()

SURFACE_BUILD *sfbuild

Compute parameter lines for a `Surface_Build` instance. This method must be called before `"merge-send"` or `"close-send"` because it needs the `geometry` structure created from the `build_parm` data to be able to find the correct points for the parameter lines.

7.4.5 Explicit

The class `Explicit` is a subclass of `Surface` for working with time-dependent surfaces given by explicit formulas (for time-dependent curves `Time_Curve` 7.4.2 should be used).

Most of the `Explicit` methods extend the superclass methods. The memory management and archiving methods just handle the new instance variables of `Explicit`, almost all other methods are needed for handling the time-dependent domain of the surface and for computing the surface from the explicit formulas.

For more information see the documentation of the associated project *Explicit* in section 8.2.5.

7.4.5.1 The Explicit Structure

`Explicit` only adds three pointers to coordinate functions and information about the time-dependent domain of the surface to its superclass.

```
#CLASS Explicit : Surface {
  DOMAIN_PARM *domain_parm;  /* Domain parameter for surface (see below) */
  COMPLEX (*compute_arg)();  /* Function pointer for user
                                supplied compute function */
  struct {
    t_fun *x, *y, *z;        /* Explicit coordinate functions  */
  } coord_f;
};
```

The data for `domain_parm` and `coord_f` are usually read from a description file with extension `*.ex` (see 8.2.5.1), the functions are read as strings, compiled into a `t_fun` with the function compiler and evaluated during the computation (refer to section 10.2.8 for a description of the function compiler). The `time_parm` variables can be used for the functions, see section 8.2.2.1.

The `compute_arg` function can be used to transform the rectangular default domain, see section 8.2.5.2 for an example.

7.4.5.2 Domain_Parm

The parameter for the `Explicit` domain are stored in a the `Domain_Parm` instance `domain_parm`:

```
#CLASS Domain_Parm : Root {
  /* PRIVATE */
  DOMAIN_PARM_RUL rul;             /* ruler for the domain parameter */
  /* USER */
  int num_parts, max_parts;        /* number and maximum number of parts */
  int *flag;                       /* flag for part orientation */
  int *num_u_lines, *num_v_lines;  /* number of lines in u/v direction */
```

```
  int *num_u_sub, *num_v_sub;      /* subdivision in u/v direction */
  SPLINE1D **u_dstrbn, **v_dstrbn; /* line distributions for each part */
  t_fun **u_min, **v_min, **u_max, **v_max;          /* domain functions */
  double **u_min_d, **v_min_d, **u_max_d, **v_max_d; /* domain variables */
};
```

Usually you don't have to access any of the `domain_parm` variables, if you want to program your own `compute_arg` function or `"compute-explicit"` method you should only use the number of parts (`num_parts`) and the number of lines (`num_u/v_lines`) and subdivision (`num_u/v_sub`) in u– and v–direction.

There are only a few methods on `Domain_Parm` for memory management and (de)activating the rulers for the minimum/maximum values:

GRAPE(dparm, "free")()

DOMAIN_PARM *dparm

Delete an instance of class `Domain_Parm`. All memory if freed using `"list-free"`, the rulers are removed from the manager and deleted. See 5.4.4.1 for details on `"delete"`/`"free"`.

GRAPE(dparm, "list-alloc")()
GRAPE(dparm, "list-free")()

DOMAIN_PARM *dparm

`"list-alloc"` allocates memory for `max_parts` domain parts, `max_parts` therefore has to be set before `"list-alloc"` is called.

The rulers, `t_fun`s and `Spline`s are not created by `"list-alloc"`, this is usually done by the function `g_domain_parm_get_data` (see below) which is called by `"read-data"` while a description file is read. `"list-free"` removes the rulers from the manager and deletes them before freeing the arrays.

GRAPE(dparm, "set")(state)
GRAPE(dparm, "switch-send")()
GRAPE(explicit, "switch-comp-send")()

DOMAIN_PARM *dparm
int state
EXPLICIT *explicit

`"set"` activates (`state == ON`) or deactivates (`state == OFF`) all `dparm` rulers, `"switch-send"` toggles their state. `"switch-comp-send"` is the `Explicit` counterpart of `"switch-send"`, it just calls this method on `explicit->domain_parm`.

There is a utility function for reading the `Domain_Parm` variables from a description file (see 8.2.5.1 for an example file):

```
int g_domain_parm_get_data(FILE *fp, EXPLICIT *explicit,
                           char *name, char *path, double *time)
```

reads in the number and maximum number of parts and allocates memory for them. Then for each part the `flag`, the number of lines and subdivisions for the u- and v-direction and the formulas for the time-dependent minimum and maximum values of the part are read. These formulas are compiled and evaluated to create splines for the rulers corresponding to the variables. The rulers are added to the manager, their default action is set to `"parameter-changed"` (see 7.4.1.5) therefore changing a ruler will recompute the surface for the new values.

7.4.5.3 Memory Methods on `Explicit`

(EXPLICIT *) GRAPE(Explicit, "new-instance")(name)

CLASS *Explicit
char *name

Create an instance of class `Explicit` with name `name` by calling `"new-instance"` on the superclass and creating the `Domain_Parm` instance `domain_parm`.

GRAPE(explicit, "free")()

EXPLICIT *explicit

Delete a `Explicit` instance. All memory is freed by calling `"list-free"`, deleting `domain_parm` and calling `"free"` on the superclass. See 5.4.4.1 for details on `"delete"`/`"free"`.

GRAPE(explicit, "list-alloc")()
GRAPE(explicit, "list-free")()

EXPLICIT *explicit

`"list-alloc"` computes the total number of points and boundaries needed for all `explicit` surfaces patches from the `domain_parm` data — which has to be initialized before `"list-alloc"` is called — and then calls the `"list-alloc"` method on the superclass `Surface`. `"list-free"` just calls `"list-free"` on the superclass and on `domain_parm`.

In contrast to `Surface_Build` where each surface patch is stored in a separate `Chain object` all `Explicit` patches are stored in a single instance.

7.4.5.4 Compute Methods on `Explicit`

GRAPE(explicit, "compute-send")()

EXPLICIT *explicit

This is the main compute method for `Explicit` instances. It allocates new memory for the `Geom2d` instances the geometry is stored in if necessary, creates the connectivity for them and computes the `domain` and the `geometry`.

The default domain is rectangular, but you can supply you own domain transformation by setting a `compute_arg` function with

GRAPE(explicit, "set-compute-arg")()

EXPLICIT *explicit,

see section 8.2.5.2 for further explanations and for an example of a user-defined domain transformation.

To compute the surface point corresponding to the domain point the method

GRAPE(explicit, "compute-explicit")(geom, num, z)

EXPLICIT *explicit
GEOM2D *geom
int num
COMPLEX z

is called with the domain point as third parameter. Normally the surface point is computed by evaluating the three `coord_f` functions and then stored as `num`th point in the `geom` instance.

You can replace this method with your own `"compute-explicit"` method if you want to compute something special which can't be described by the formulas in the input file, please see section 8.2.5.3 for a detailed documentation and an example.

In contrast to `Time_Object` you should never replace `"compute-send"` to compute your own `Explicit` surfaces but supply a `compute_arg` function or `"compute-explicit"` method instead.

7.4.5.5 I/O Methods on Explicit

GRAPE(explicit, "read-data")(path)
GRAPE(explicit, "get-extension")()

EXPLICIT *explicit
char *path

"read-data" reads in the Explicit description file path/name.ex where name is the name of the explicit instance. *.ex is the default extension for all Explicit input files, it is returned by "get-extension". "read-data" uses the function g_domain_parm_get_data to read the domain_parm data from the file, it creates rulers for the minimum and maximum values of the parts and for the parameter line distributions and adds them to the manager.

GRAPE(explicit, "xdr")(xdrp)

EXPLICIT *explicit
XDR *xdrp

Read the explicit instance from or write it to the XDR stream xdrp. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

7.4.5.6 Other Methods on Explicit

GRAPE(explicit, "merge-send")()

EXPLICIT *explicit

Additionally to merging all geometries by calling the superclass method (7.4.3.8) all explicit rulers are deactivated — recomputing an Explicit instance after merging it is not possible.

GRAPE(explicit, "assign-cons-send")()

EXPLICIT *explicit

This method computes the parameter lines in u- and v-direction for the explicit instance and stores them in the hor and ver instances of the Geom2d instances of the explicit geometry. "assign-cons-send" must be called before "merge-send" or "close-send" because it needs the geometry structure created by "compute-send" to be able to find the correct points for the lines.

If this method was called accidentally and you don't need the parameter lines them you should delete them with "delete-lines-send", this will save a lot of memory and make some methods, especially "merge-send" and "close-send", much faster.

GRAPE(explicit, "set-task")(task)

EXPLICIT *explicit
TASK_TYPE task

Set the explicit->task to task. Additional to the superclass method (7.4.3.6) the tskDomain and tskExplicit (which changes the model to the geometry like tskGeometry and corresponds to the EXPLICIT task in the input file) tasks are handled.

7.4.6 Amandus

Amandus is a subclass of Explicit which is designed to work with minimal surfaces defined via the Weierstrass representation formula. This is (currently) the only subclass of Surface which requires programming: the functions for the Weierstrass representation and for computing the domain of the surface have to be provided as C functions, for details see section 8.2.6 about the *Amandus* project.

7.4.6.1 The Amandus Structure

The class `Amandus` adds an additional geometry `newton` which is used to store the result of the analytic continuation of the surface (computed with a complex Newton iteration) and the functions needed for the Weierstrass representation to its superclass. The `weier_f` function pointers are initialized by the `"get-weier-f"` method (see below) which is called by `"read-data"` when an `Amandus` description file is read. The two `CNODE` structures are used to store some data during the integration, they should only be used internally.

```
#CLASS Amandus : Explicit {
  CHAIN *newton;
  struct {
    int type;
    COMPLEX (*x)(COMPLEX), (*y)(COMPLEX), (*z)(COMPLEX);
    COMPLEX (*g)(), (*dh)(), (*dg)();
    COMPLEX (*poly)(COMPLEX, COMPLEX);
    COMPLEX (*dwpoly)(COMPLEX, COMPLEX);
    COMPLEX (*dzpoly)(COMPLEX, COMPLEX);
    COMPLEX (*newton_init)(COMPLEX);
    void (*correct)();
  } weier_f;
  CNODE *cnode, *icnode;
};
```

7.4.6.2 Memory Methods on Amandus

(AMANDUS *) GRAPE(Amandus, "new-instance")(name)

CLASS *Amandus
char *name

Create an `Amandus` instance with name `name`. First `"new-instance"` on the superclass `Explicit` is called and then the `CNODE`s needed for the computation are allocated. The memory for the geometries has to be allocated by calling `"list-alloc"`, the `weier_f` function pointers are initialized by `"get-weier-f"` which is called when an `Amandus` description file is read.

GRAPE(amandus, "free")()

AMANDUS *amandus

Delete an instance of class `Amandus`. First `"list-free"` is called and the `CNODE`s are deleted, then `"free"` on the superclass is called. See 5.4.4.1 for more information on `"delete"`/`"free"`.

GRAPE(amandus, "list-alloc")()
GRAPE(amandus, "list-free")()

AMANDUS *amandus

Both methods extend the superclass methods, they allocate/free the `newton` geometry for an `Amandus` instance. `"list-alloc"` creates `num_of_frames` frames for `newton`, the `domain_parm` values determine how many points are allocated for each frame.

7.4.6.3 Display Methods on Amandus

GRAPE(amandus, "newton-disp")()

AMANDUS *Amandus

Display the time section of `newton` for time `time`. The time section is created by calling `"get-object"(*time)` and stored in `inter` which then is displayed.

GRAPE(amandus, "gausscurv-disp")()

AMANDUS *amandus

This method only works if the local representation `KARL` is used and if the task is set to `tskGeometry`, it shows isolines for the Gaussian curvature

$$-\left(\frac{2}{|G|+1/|G|}\right)^4 \frac{|dG/G|^2}{|dh|^2}$$

using a linear FE function (the formula is evaluated at each vertex).

7.4.6.4 Compute Methods on Amandus

Computing a minimal surface via the Weierstrass representation formula is a very complex task. First the parameter domain is created, then the Gauss map (using analytic continuation is necessary) and finally the surface itself are computed. This is done with the methods described in this section, for more information see 8.2.6.

GRAPE(amandus, "compute-send")()

AMANDUS *amandus

This is the main compute routine for `Amandus` instances. Depending on the task `domain`, `newton`, `gauss` and `surface/conjugate` are computed, see `"compute-node"` below. All computed points are drawn immediately. By supplying a user defined `correct` function numerical inaccuracies, problems with branch points etc., can be corrected after the computation.

GRAPE(amandus, "compute-node")(old, new)

AMANDUS *amandus
CNODE *old, *new

Compute a `new` point from a given `old` one. Depending on the task first `domain` and then `newton` (if necessary), `gauss` and `surface/conjugate` are computed with the following methods:

GRAPE(amandus, "compute-domain")(new)

AMANDUS *amandus
CNODE *new

Add a point to the parameter domain of a minimal surface. The `new` point is computed by the `"compute-send"` method, if no user `compute_arg` function is set a rectangular grid is used. See section 8.2.5.2 for further explanations and an example.

GRAPE(amandus, "compute-newton")(old, new)

AMANDUS *amandus
CNODE *old, *new

Analytic continuation of `old->w` from `old->z` to `new->z` with `NUM_OF_STEPS - 1` intermediate points. The points are stored in `NEWTON_VEC_Z`, the computed values in `NEWTON_VEC_W`. The value computed for the endpoint `new->z` is stored in the `newton` geometry.

The main computation routine (a complex Newton iteration) is

```
COMPLEX cnewton(COMPLEX zold, COMPLEX wold, COMPLEX znew,
                COMPLEX (*f)(), COMPLEX (*dwf)(), COMPLEX (*dzf)())
```

`dzf` may be `NULL`, but the results are much better if this function is supplied.

The values computed with `"compute-newton"` are used by

```
COMPLEX get_newton(COMPLEX z)
```

to compute `w` for values of `z` in the interval from `old->z` to `new->z`, it searches the values stored in `NEWTON_VEC_W` for an initial value and applies a further newton iteration if necessary. This function is for example called by the complex Romberg integration routines that are used by `"compute-surface"` to integrate the Weierstrass data.

GRAPE(amandus, "compute-gauss")(new)

AMANDUS *amandus
CNODE *new

Compute of image of the `domain` points of the Gauss map by evaluating the `weier_f->g` function.

GRAPE(amandus, "compute-surface")(old, new)

AMANDUS *amandus
CNODE *old, *new

Compute the points and the normals (stereographic projection of the Gauss map) of the minimal surface and its conjugate surface. This method is called after the other compute methods, it integrates the Weierstrass data from `old` to `new` using complex Romberg integration.

7.4.6.5 I/O Methods on `Amandus`

GRAPE(amandus, "read-data")(path)
(char *) GRAPE(amandus, "get-extension")()

AMANDUS *amandus
char *path

These methods are used by the `read *.in` button's method `"read-in"`, see 7.4.1.7. `"read-data"` reads in the description file for class `Amandus` with name `path/name.am`, it allocates the memory needed for the surface described in the input file and initializes the `weier_f` function pointer by calling `"get-weier-f"`. `"get-extension"` returns the extension `*.am` for `Amandus` description files, it is used by the file requester poped up by `"read-in"` to determine which files should be shown.

GRAPE(amandus, "read")()

AMANDUS *amandus

Read an `Amandus` instance from the standard textdevice. This method is only provided for backward compatibility to read files written with `"write"`, it is replaced by

GRAPE(amandus, "xdr")(xdrp)

AMANDUS *amandus
XDR *xdrp

Read the instance `amandus` from or write it to the XDR stream `xdrp`. This method should not be called directly, refer to 5.7 (especially 5.7.3) for details.

7.4.6.6 Other Methods on `Amandus`

GRAPE(amandus, "get-weier-f")()
GRAPE(amandus, "set-weier-f")()

AMANDUS *amandus

This method is called by `"read-data"` when an `Amandus` description file is read, it initializes the `weier_f` function pointers. The `name` of the instance `amandus` is used to determine which functions should be used, if no appropriate functions are found the method `"set-weier-f"` is called. The user may provide its own `"set-weier-f"` method to add `Amandus` surfaces, see 8.2.6.

GRAPE(amandus, "assign-cons-send")()

AMANDUS *amandus

This method computes the parameter lines in u- and v-direction for the `amandus` instance and stores them in the `hor` and `ver` instances to the `Geom2d` instances of the geometries. This method has to be provided since the connectivity created by `"compute-send"` on `Amandus` differs from the connectivity used for `Explicit` instances. This method only works for the original surface patch, i.e. it must be called before `"merge-send"` or `"close-send"`.

GRAPE(amandus, "merge-send")()
GRAPE(amandus, "close-send")()
GRAPE(amandus, "insert-step-send")()
GRAPE(amandus, "reset-send")()
GRAPE(amandus, "set-task")(task)

AMANDUS *amandus
TASK_TYPE task

These methods extend the corresponding methods on `Surface` described in section 7.4.3.8, the additionally handle the `newton` geometry resp. the `tskNewton` task.

Chapter 8

Projects

This chapter contains a detailed description of the predefined **GRAPE** projects. For each project you will find a description of its functionality, how to use its interface and on which classes and methods it is based. Some of the project descriptions also contain examples which further explain their use. A general description of what projects are and how they are used can be found in sections 4.5 and 5.6.

Projects can depend on each other, they form a hierarchy which is shown in figure 8.1. In contrast to the class hierarchy there is nothing like a base class, a project may depend on several other projects. Since a project loads the projects it depends upon you should additionally to the documentation of the project itself read the documentation of the projects it depends upon.

Some projects just combine other projects to a larger package or provide an interface to methods from the GRAPE library. In these cases you will find references to the appropriate parts of the documentation.

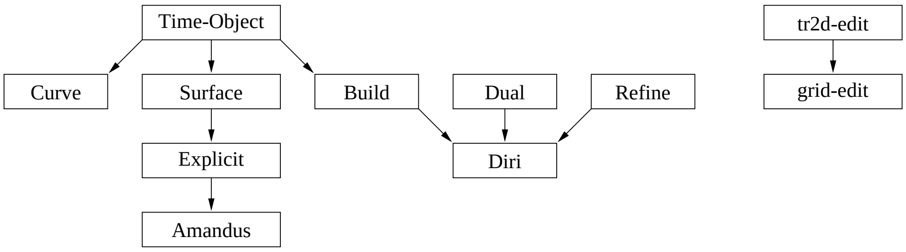


Figure 8.1: Project Dependencies

8.1 Projects for Continuum-Mechanics

8.1.1 Project Move-Trace

After adding the project move–trace several buttons and rulers to handle the moving and tracing are provided in the options menu **opt1**.

To start or continue one of the the moving methods for points, objects or streaks on the current scene (with a vector valued Fe2d or Fe3d object) press one of the buttons **start** or **cont**. A detailed description of these methods can be found in chapter 7.2.2. To **move objects** it is neccesary that a test set (a surface (Triang2d) or curve (Triang1d)) is located at the position of the next scene.

With **create** in the **handle polygons layer** a polygon can be generated by picking the points on a clipping plane. **Refine** in the same layer on an polygon refines it by dividing each interval in half (adding a new point). The parameter **refine (% pts.)** adjustes the maximal number of new points in each moving step (**number of steps**). Be careful with large values in combination with

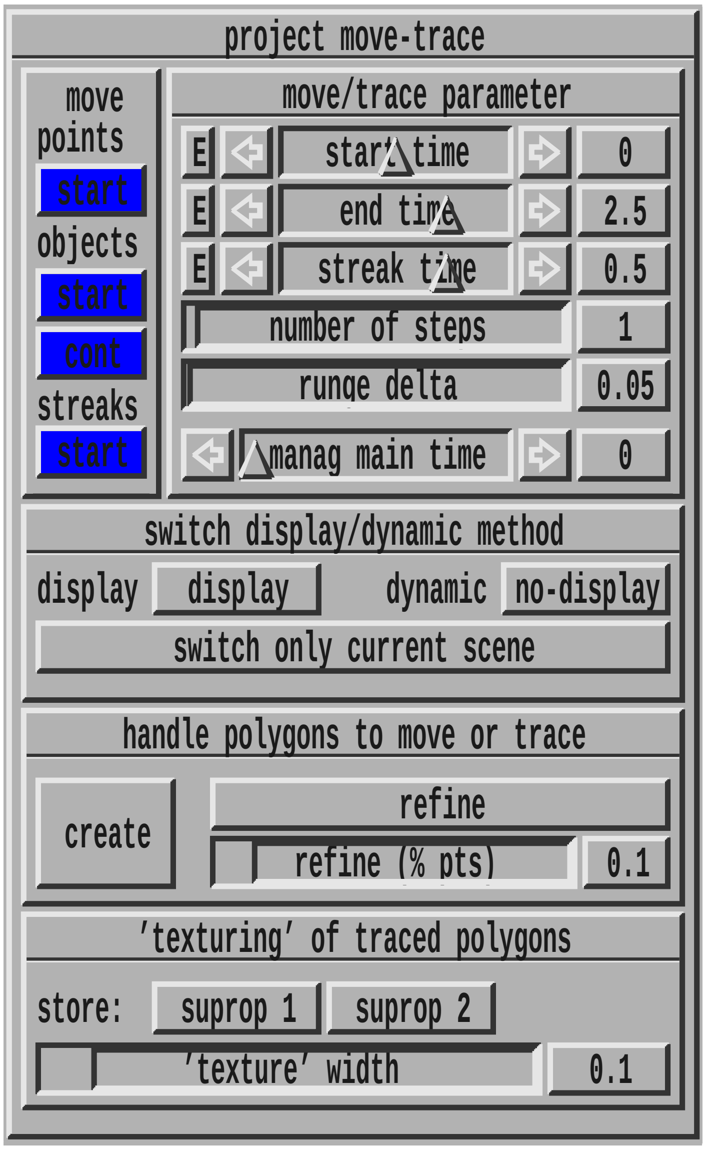


Figure 8.2: The layout of the project move–trace

large number of steps because of the the cost of memory. Another possibility to get an object to be moved is to read it from disk using **read gr** in the manager menu.

The bf move/trace paramter can be adjusted by using the ruler **start time**, **end time**, **streak time**, **runge delta**, and **number of steps**. There the two time rulers are multiple used:

- to determine the **start** and the **end time** used in the moving methods for points, objects and streaks
- as **start** and **end time** of the two display methods **tracing** on Trace and Timestep (if the objects in the sequence of timesteps are of a subclass of Triang1d)

The other three rulers are only used to determine the moving parameters described in chapter 7.2.2, too.

The buttons in the **switch display/dynamic** layer alter the **display**– rather the **dynamic**–display–method of the moving objects. The **display** button shifts the display method of all timescenes stringed together between display and **NULL** round. The **dynamic** button shifts the dynamic display method of all timescenes stringed together between **NULL**, **display** and **tracing** round. The method tracing on timestep (with Triang1d objects) has an ability which uses the two texture rulers and the two suprop buttons additional to the time rulers. With the buttons **store suprop 1** and **store suprop 2** two different surface properties can be stored like the button **suprop** in the manager menu does. The two surface properties are used to display the rendered moving curve (see Tracing methods in chapter 7.2.3) as a surface consisting of ribbons with these surfaces properties. The width of these ribbons can be modified by varying the ruler **'texture' width**.

8.1.2 The Grid Editing Projects

Now we turn to a detailed description of the functionality of the previously mentioned grid editing projects. The two can be added by pressing the button **add project** in the manager of GRAPE. The buttons and rulers of these projects are inserted in the options menu **opt1**.

8.1.2.1 The Project tr2d–edit

Basic Modifications on a Given Mesh

The project **tr2d–edit** includes some basic tools for mouse oriented mesh building. All tools of this project work on objects of the class TRIANG2D, which is the basic one for 2D simplicial meshes. Some basic features are:

- adding new triangles
- removing triangles
- changing the position of the gridpoints
- removing gridpoints.

Pressing the button **del triangle** causes every triangle that is first marked by picking with the left mouse button to be erased after the middle mouse button is pressed. The marking of a triangle can be canceled by a second picking on this triangle. After selecting **add triangle**, we can choose three points with the left mouse button. A new triangle will be created using these three points and the data structure of the triangulation will be conformed. The points can be nodes of the mesh or new points. However, this method should not be used to refine a triangle. The button **modify point** has the effect that we can pick a node of the mesh and mark its new position with a second click on the left mouse button. With the middle mouse button the new position will be accepted and the data structure will be updated. To select points, we can use the utility **grid for picking** by pressing the corresponding button. With this tool the user can select background gridpoints. If no grid for picking is chooser we can select existing mesh points in an user defined epsilon circle. The radius epsilon can be varied with the **picking accuracy** slider. After pressing the button **grid for picking** we can choose with the middle mouse button several grid modes which differ in the fineness of the grid for picking. With the left mouse button we can select a node of the current triangulation and align it to the grid for picking.

After modification of the triangulation it is recommended to verify its consistency. This can be done by using **check**. Thereafter a warning will be displayed if the vertex field, neighbour field, connectivity, Euler–Poincare–Characteristic, boundary or positive volume of the generated triangulation is violated.

Refinement Tools

The following two refining techniques for the class TRIANG2D are included in the project **tr2d–edit**:

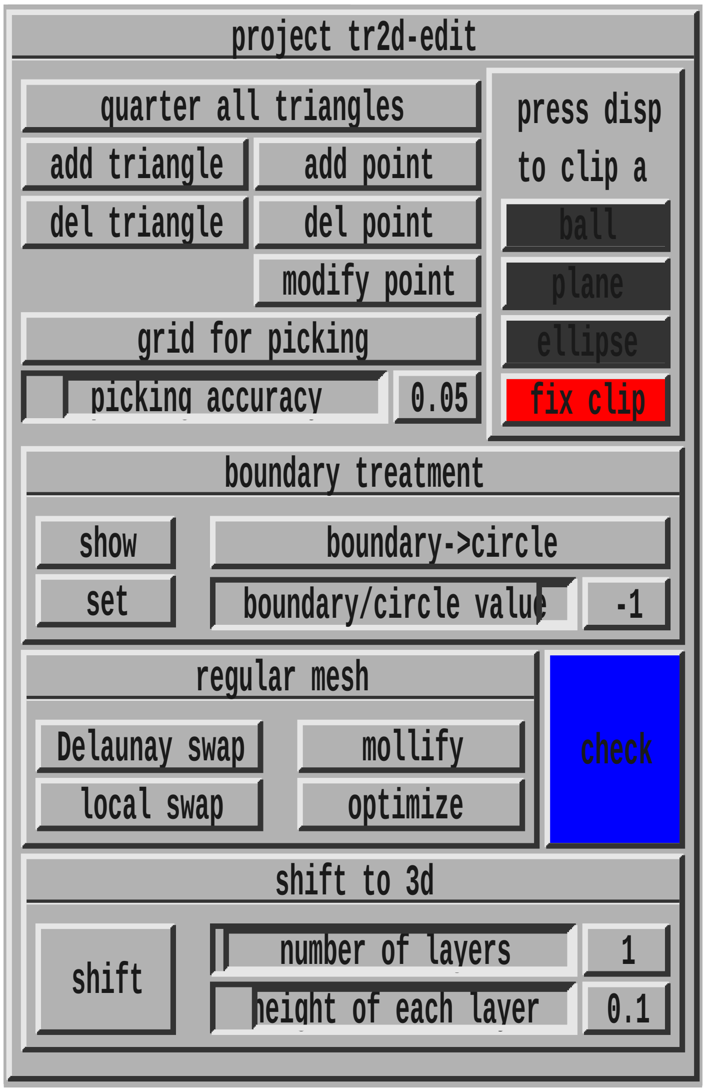
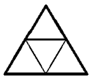
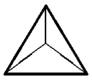


Figure 8.3: The layout of the project tr2d–edit

- regular refinement (quartering of the triangles) 
- refinement with regard to the center of gravity  .

The global refine technique which divides all triangles of the grid into four triangles can be initiated by clicking the button **quarter all triangles**. The local refine technique which divides a triangle into three triangles can be choosen with the button **refine center**. With the left mouse button we can mark triangles for refining by picking. Again, the marking of a triangle can be canceled with a second picking. With the middle mouse button the refining of the marked triangles is performed.

Instead of the impossible direct reversion of these operations, the point oriented coarsening technique can be chooser with the button **del point**. Thereby each point of the triangulation can be removed. To get a conforming mesh some triangles must be removed or added in addition. As an example we refine some triangles with regard to the center of gravity in a uniform mesh (cf.

Fig. 8.4, left) and make the grid more regular by swapping the edges (cf. Fig. 8.4, middle). Then we remove some points in the refined domain and get a mesh similar to the uniform initial grid (cf. Fig. 8.4, right).

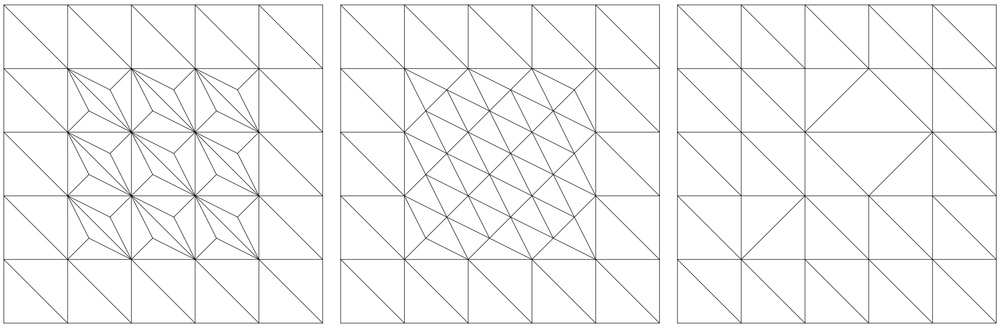


Figure 8.4: Uniform mesh after refining with regard to the center in the middle of the domain, swapping and point oriented coarsening

We can select the grid node which should be erased with the left mouse button. With the middle mouse button we can start this process.

Modification of the Boundary Values

To identify the mesh border, the *neighbour* values in the TRIANG2D data of boundary triangles are set to a negative value, usually -1. To distinguish between different types of boundaries, especially in a numerical algorithm, we can use different negative values. This can be done pressing the button **set** in the **boundary treatment layer**. With the left mouse button we have to pick the first and the last boundary edge counterclockwise (clockwise) for outer (inner) boundaries we want to modify. Pressing the middle mouse button we give the selected boundary part the value which has previously been tuned by shifting with the ruler **boundary/circle value**.

Using **show** the boundary segment with a boundary value corresponding to the ruler **boundary/circle value** are shown.

Surface Clipping

Some of the popular tools in GRAPE are the clipping methods. On 2D triangulations we can distinguish between two types of clippings:

- the display methods which do not change the current object, in our case a 2D mesh
- the fixing methods which change the current object according to the parameters adjusted for the display method.

The three display methods in our project are the surface clipping of a ball, a plane or an ellipse. We can change the current display method to one of these three clipping methods by pressing the button **disp** followed by a clicking on one of the clipping buttons **ball**, **plane** or **ellipse**. Then the rulers to modify the position and the e shape or direction of the clipping surface are added in **opt0**. In the case of a ball or an ellipse a negative radius can be used to get the outside domain. After we have adjusted the clipping surface we press the button **fix clip** to fix the mesh in the displayed way. Be careful because this operation is not reversible. At last by pressing the **disp** button twice we return to the standard display method.

Another possibility to clip a circle out of a 2D mesh is to use the method **boundary->circle**. First we have to erase several inner triangles which build a coarse approximation of the circle (cf. Fig. 8.5, middle). Then we have to give the new boundary triangles an unused boundary value. For this we can use the button **set** in the **boundary treatment layer** (see section 8.1.2.1). After clicking the button **boundary->circle**, the boundary points marked with the value of the ruler **boundary/circle value** will be moved onto a circle (cf. Fig. 8.5, right). The value of the circle radius is chooser as the average of the distances between the boundary points and the center of the cutted domain.

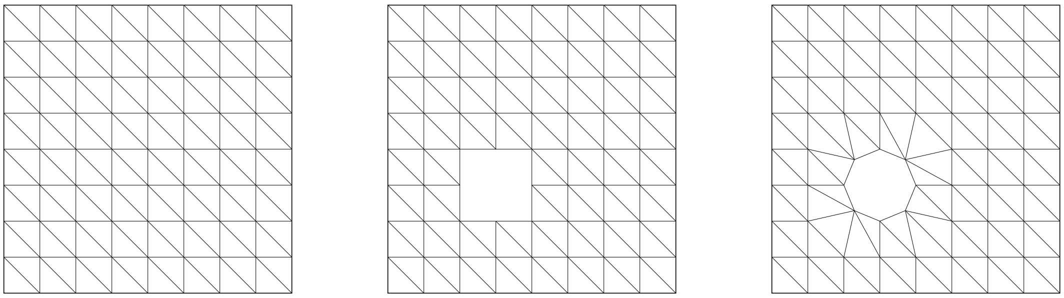


Figure 8.5: Another technique to cut a circle out of the mesh

It is possible to refine the mesh or only some triangles and to press the button **boundary ->circle** again (cf. Fig. 8.6).

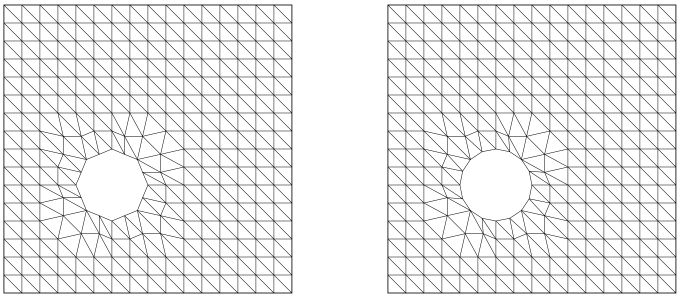


Figure 8.6: After refinement the boundary points can moved onto the circle again

The Handling of the Mesh Optimizers

The handling of the methods to optimize the grid (make it more regular) will be demonstrated in this section. For swapping edges we click the button **Delaunay swap**. Then all edges of the mesh will be swapped until no edge fulfills the swap criterion. If we want to swap an edge although the criterion is not fulfilled we have to press the button **local swap**. Then we pick the edges we want to swap with the left mouse button. With the right mouse button we can exit this method. For using the method *mollify* we only have to select the button **mollify**. But remember, one click on this button is only one iteration step. Similar, *mesh–optimize* can simply be executed by pressing the button **optimize**.

8.1.2.2 The Project grid–edit

The project **grid–edit** was developed for ADAPT2D data. All additional methods which were collected in this project only work on ADAPT2D. The basic differences between ADAPT2D and TRIANG2D are the REFINE2D structure for each triangle and the structure BND_FUNCTION. With information of the REFINE2D structure the mesh can be coarsened after refinement. The structure BND_FUNCTION allows to store information about the boundary curves.

The Handling of the Boundary Curves

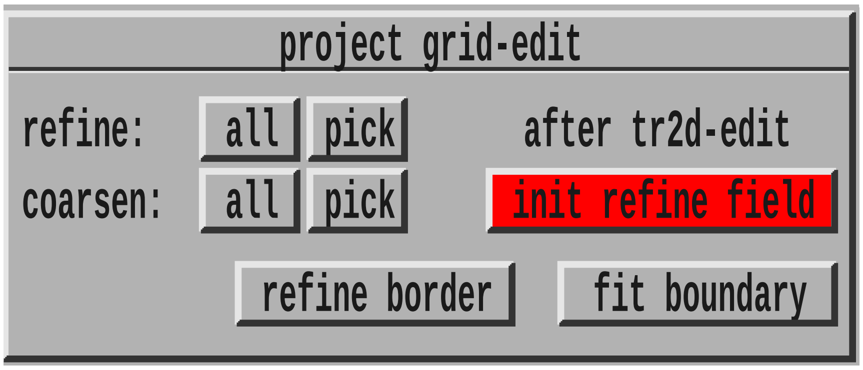


Figure 8.7: The layout of the project grid–edit (without tr2d–edit)

For meshes whose boundaries can be described in an implicit form their shape is stored in the structure

```
typedef struct bnd_function {
        double (*g)();
        void *par;
} BND_FUNCTION; .
```

Where *g(x,y,z,bnd_function→par)* $= 0.0$ is the implicit function describing the boundary curve with parameter *par*.

For instance to get a ball, the function g and parameter *par* are defined as follows:

```
typedef struct ball_parm_2_type {double cx,cy,cz,r;} BALL_PARM_2;
static double ball(x,y,z,var)
     double x,y,z;
     BALL_PARM_2 *var;
{ double ret;
    ret = (x-var->cx)*(x-var->cx)
        + (y-var->cy)*(y-var->cy)
        + (z-var->cz)*(z-var->cz) - (var->r*var->r);
    if (var->r < 0.0) ret = - ret;
    return(ret);
} .
```

The function `ball` returns the distance between the 3D point `(x,y,z)` and the ball with its center `(var->cx,var->cy,var->cz)` and radius `var->r`. If the point `(x,y,z)` lies on the ball the function returns the value zero and if the point lies inside (outside) the ball, a negative (positive) value will be returned.

In GRAPE boundary curves describing a ball, ellipse or plane are supported by appropriate predefined functions.

The boundary nodes which belong to a boundary curve stored in the structure BND_FUNCTION can be projected on the boundary curve with the method *fit_bndfunctions* at any time to this boundary curve. It can be activated by pressing the button **fit-boundary**. Thereafter the nodes on all boundary curves stored in boundary functions are moved onto the corresponding boundary curves.

The Handling of the Mesh Adaption Tools

The grid adaption (refining by dividing in half, coarsening) on an ADAPT2D triangulation can be done in a global or local manner. The two global methods with the corresponding buttons

refine all/coarsen refine/coarsen the whole mesh. After pressing the button **refine all** all triangles of the mesh are divided in half with regard to their refinement edges. With **coarsen all** the simply refined situations are coarsened. Mostly we have to press this button two or three times to dissolve back to the starting triangulation. With the methods **refine pick** and **coarsen pick** we can pick the triangle we want to refine or to coarsen. These local operations are carried out immediately.

To refine only a border segment, instead of using **refine pick** and picking all triangles along the boundary, the button **ref border** can be activated. It works similarly to **set** in the **boundary treatment layer**. We have to mark the boundary segment with the left mouse button. With the middle mouse button the marked segment will be refined applying the refinement technique described above.

The large red button **init refine field** is very important.

After all grid modifications:

- clipping
- removing triangles or points
- adding triangles
- swapping
- quartering
- refining with regard to the center of gravity

it is always necessary that the button **init refine field** is pressed before we use any refine/coarsening method on an ADAPT2D mesh. The reason is obvious: the REFINE2D structure in ADAPT-2D is not influenced by any of the above mentioned methods which all work on the class TRIANG2D. Therefore additional information stored in REFINE2D does not fit (concerning the enumeration) to the TRIANG2D structure. Do not forget to press **init refine field** but remember after pressing it, it is no longer possible to coarsen any refined triangle.

8.1.2.3 An Example for Generating a Grid

After starting the interactive package and adding the project **grid–edit** we switch to the first options menu **opt1** and start with a macro triangulation of the square domain, consisting of two triangles (cf. Fig. 8.8). This can be generated using **add triangle** with a **background grid** or one of the GRAPE methods *explicit* or *mask-make* sent to class ADAPT2D. After refining this grid (cf. Fig. 8.8) with **quarter all triangles** (one of the three possibilities to refine a grid) we want to clip out the first ellipse.

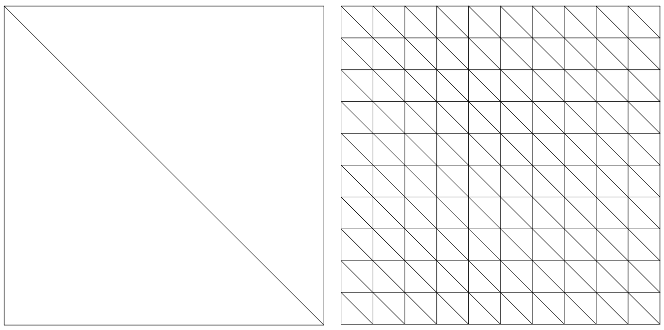


Figure 8.8: The starting triangulation and the refined one

To do so we change the current display–method by pressing the button **disp**. Then the caption of this button changes to **disp ->** and we can choose **ellipse**, the display–method we have to take into account here. In **opt0**, where all the rulers of display–methods are added, there are five new rulers to adjust the position, length and height of the ellipse. To fix the changes of our mesh we have to press **fix clip**. The ellipse is stored as a boundary function only when **clip-fix** is used on an object of class ADAPT2D. Then we can change the display–method back to the standard display by pressing the **disp** button twice. The result is shown on the left hand side of Figure 8.9.

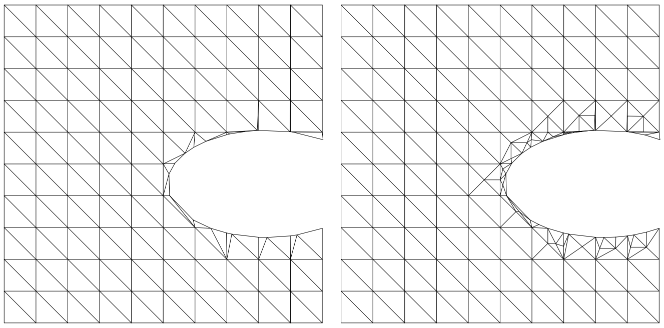


Figure 8.9: After clipping out an ellipse and boundary refinement

Now we refine the boundary along the ellipse to get a smoother curve. At first we have to press the button **init refine field** to initialize the information in the ADAPT2D structure which is misleading after clipping, removing, adding, modifying and swapping. Then we can use **ref border** and mark the first and last segment (counterclockwise with the left mouse button) of the boundary we want to refine. After the refinement is performed (cf. Fig. 8.9) the new nodes on the mesh boundary were automatically pushed onto the boundary curve of the ellipse. Next we optimize the mesh by calling the methods **mollify** or **optimize** several times (cf. Fig. 8.10).

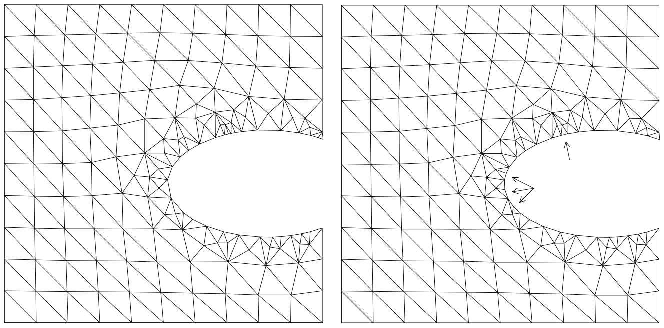


Figure 8.10: With mesh optimization and some handwork

In the areas where we are not content with the triangulation we refine or coarse some triangles by hand using **refine pick** or **coarsen pick** (cf. Fig. 8.10). Now we apply the same procedure to a second ellipse (cf. Fig. 8.11), adjusting the parameters and fixing the geometry as described above.

Next we try to make the actual triangulation (cf. Fig. 8.12, left) more regular and uniform along the boundary. To do this we at first use **Delaunay swap** and then remove the points which are not needed by picking them after pressing **del point**. The result shown on in the right hand side of Figure 8.12 seems to be very convincing concerning the smoothness along the border and the angles in the triangles and forms a good starting mesh for a numerical algorithm.

8.1.3 Project Probe

This project offers several tools for easy data-analyzing. After adding the project *probe* three buttons and a ruler will occur in the first optionsmenu. The buttons allow to start the methods

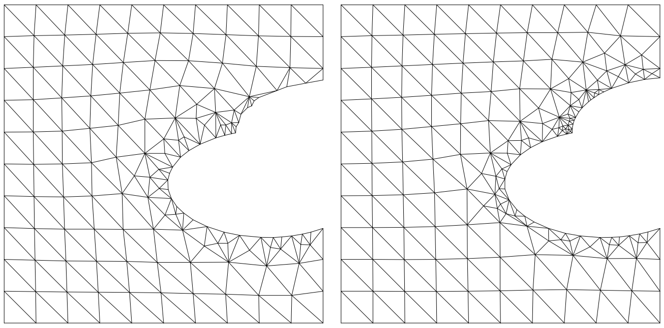


Figure 8.11: The same procedure with a second ellipse

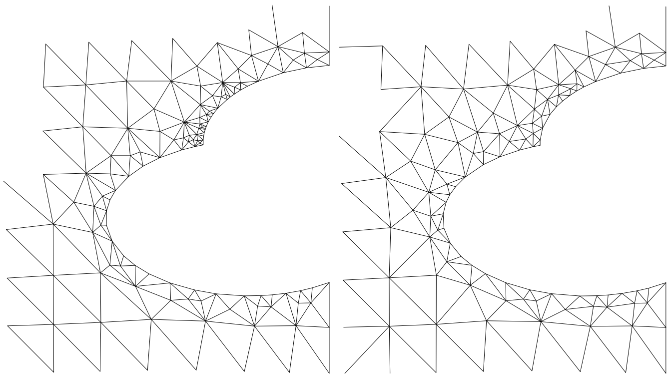


Figure 8.12: A zoom to show the 'quality' of the grid along the boundary before and after using **del point** and **Delaunay swap**

point-probe, *line-probe* and *curve-probe*, working on timedependent or timeindependent data. Using point-probe or line-probe you can pick on your data. Line-probe will divide a so marked line in as many parts as the number of the ruler shows. Curve-probe looks for the next triang1d-object in the scene-tree and works like line-probe. In the timedependent case all methods open a new graphic-window. Point-probe plots a time-data-graph, both other methods plot a time-arc-length-data-graph. In the timeindependent case point-probe writes the data at a picked point in the manager-window. Line- or curve-probe plot a arc-length-data-graph. The displayed data is saved in files named point.tab or curve.tab. They can be processed by software like gnuplot.

8.2 Projects for Differential Geometry

The projects from *Time-Object* (8.2.1) upto *Amandus* (8.2.6) correspond to the classes described in sections 7.4.1–7.4.6 where all methods are documented. Here we only describe their interfaces and how to use these projects.

8.2.1 Project *Time-Object*

This project provides the basic environment for working with time-dependent objects, most projects described in the following sections are based on it. Adding project *Time-Object* or one of its subproject will install a user-interface in the option menu and add an object of the corresponding class to the data hierarchy.

Since description files are used to define parameter, domains, surface patches etc. most of these projects don't require any programming. There are two exceptions: creating *Time-Objects* requires to write a `"compute-send"` method and for *Amandus* the functions for the Weierstrass representation formula have to be written as C functions.

The geometry of a `Time_Object` is arbitrary (its class only has to define a `"get-object"` method) therefore the user has to provide the `"compute-send"` method (compare 7.4.1.5) to allocate the memory for it and to create the data or to read it in. An example for a description file and the corresponding `"compute-send"` method can be found in section 8.2.1.3.

8.2.1.1 The *Time-Object* User-Interface

The project *Time-Object* user-interface provides buttons to read description files, to navigate through `TimeSteps` and to handle `int` and time-dependent `double` parameter. Additionally it provides access to some debugging methods and to methods managing (parts of) different geometries.

The interface always shows the data of the current `Time_Object`, the title contains the class and name of this object. The interface may be partly deactivated if no object is available.

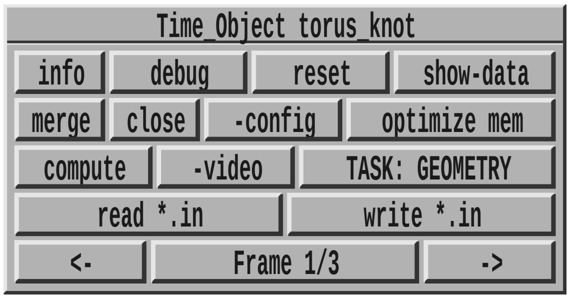


Figure 8.13: User Interface for Project Time_Object

`info` Pop up a layer showing information about the current object in the data hierarchy, pressing this button has the same effect as clicking with the left mouse button on the current object in the manag menu. What information is presented depends on the `"get-info"` method of the object's class.

`debug` Change the display to debug mode. You can now pick points or objects in the graphic window, information about the objects picked are printed to the shell **GRAPE** was started in. In debug mode the control window is deactivated, only mouse clicks in the graphics window are accepted. As usual the left and right mouse button are used to pick objects and the middle button aborts, see the message-field at the bottom of the manager window.

For `Surface` and its subclasses a `"debug"` method is provided (this is possible because the geometry structure of these classes is fixed), for `Time_Object` instances debugging only works if a method `"debug"` is defined on the class of the instance's geometry.

`reset` Reset the `Time_Object` to the initial state, i.e. as if nothing had been computed yet. This is only necessary for subclasses like `Amandus` which use a step by step approach to compute their objects.

`show-data` Pressing this button calls `"show-send"` on the current object in the data hierarchy. This methods prints lots of information to `stdout`, for `Triang` classes for example the coordinates of all points and the connectivity of all elements is printed.

Be careful when using this button on a `Time_Object` with many frames, the data of all frames will be printed....

`merge` If a `Time_Object` consists of several separate geometries (i.e. if there are several instances in its geometry chain, this is for example the case for a cube created with the `Surface_Builder` which consist of six objects (sides) or for a `Surfaces` created by reflection) they are merged into a single geometry.

`close`, `-config` After merging geometries it sometimes is necessary to remove the by now interior boundary lines of the formerly separate objects. The neighbourhood connectivity is then extended over the whole surface, applying this to curves may close start and end point if they are close together.

The close operation is effected by several parameter which can be controlled with the layer that pops up when the config button is pressed. The ruler controls upto which distance boundaries or points

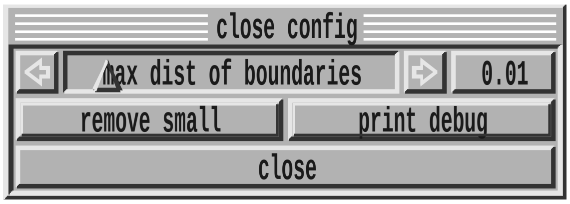


Figure 8.14: Close Config Layer

should be identified, for the comparison the ruler value weighted with the boundary lengths is used. If the checkbox `remove small` is on boundaries smaller than the (absolute) ruler value are completely removed, if `print debug` is on some debugging information is printed to `stderr`. The layer stays open until it is removed with the `close` button.

`optimize mem` This cleans up the geometries. Depending on the class of the geometry this will remove unused points, free unused memory, remove inactive curves etc.

`compute`, `-video` Compute the objects of the current frame or the whole sequence depending on the current task (see next paragraph). For some classes like `Amandus` it may be necessary to press the `reset` button to be able to recompute objects.

`TASK:...` The task controls which geometries are computed when the `compute` button is pressed. It allows to attack mathematical problems by computing an object in a step by step process, at first simple things like a parameter domain, then specific functions in that data and at the end the whole geometry. For *Time-Object* only the task `GEOMETRY` is allowed, subprojects add further tasks.

`read *.in`, `write *.in` Read or write (still not implemented) a description file. The type of the file — specified by the file extension (`*.to` for *Time-Object*, for a description of the format see section 8.2.1.3) — depends on the class of the current object in the scene hierarchy. After clicking on this button a file requester appears and you can select a file.

Remark: If a time-dependent double parameter is defined in the description file `name.extension`, the system searches the current directory for a spline description file with name `name_parm_i.sp` where `i` is the number of the variable. If this file exists it is used otherwise a default spline is created by the system, for further information see the next section.

`<-`, `Frame i/n`, `->` The button in the middle shows the current frame and the number of frames for the current `Time_Object`. With the arrow buttons you can step to the `->` next or the `<-` previous frame, by pressing the `Frame i/n` button and typing in a number you can select a frame directly.

Additionally to these buttons a `geometry displays` button is added to the option menu by this project. When it is pressed the layer shown in figure 8.15 offering some display methods appears. For `Time_Objects` only `"geometry-disp"` is available, the subclasses add further geometries and therefore more display methods.

8.2.1.2 Time and Variable Rulers

Depending on the `Time_Objects` loaded the option menu may contain several rulers for `int` and time-dependent `double` variables.

By default all `Time_Object` instances use the same `time` variable. This variable can be edited with the time ruler in the option menu or with the local (scene) time ruler in the manag menu, see section 4.3.4. Changing the time will not affect the current frame but changing the frame will set the time to the current frame's time.

For each variable defined in a description file that is read in a ruler is added to the option menu. All `double` variables are time-dependent, they are functions over the time which are defined by splines. A variable can be changed by editing the spline or by changing the ruler directly. For `double` rulers either a constant (default) spline is created or a spline is read from the disk, this is the case if a file named `name_parm_i.sp`, where `i` is the number of the variable, exists in the directory the description file `name.extension` was read from.

If you press the ruler's `E` button a spline editor pops up and you can edit the spline (see 4.1.2.10). For objects of some classes it might be necessary to change the time or even to press the `reset` and the `compute` button to see the effect of the changes.

For some subclasses of `Time_Object` changing double variables by editing the ruler directly also changes the spline, inserting new knots if necessary. For other classes changing the ruler is only possible if the current time is equal to the current frame's time, in this case the changes are local, they don't affect the spline and are lost when the time is changed and the variable is updated by evaluating the spline. Just open the spline editor, change the ruler and see what happens.

You can save the spline corresponding to the variable to make the changes permanent, the next time you read in the description file the spline file will be read automatically.

8.2.1.3 A *Time-Object* Example

The format of the *Time-Object* description files is very simple, it just contains task, frame and parameter information. All description files of subclasses of `Time_Object` contain this data, to parse it the function `g_time_parm_get_data` (see 7.4.1.2) is provided.

```
#
# torus_knot.to: test object for time_object - (m, n) torus knot
#

# task
GEOMETRY;
```

```
# frame data
0; 3;           # frame / number of frames

# parameter data
2; 2;           # number of int / double parameter

# int parameter
8;              # m
3;              # n

# double parameter
2.5;            # radius
1.0;            # torus radius
```

As you can see each number or string has to be terminated by a `;` and comments start with `#`. The task always has to be `GEOMETRY` for `Time_Object` files, subclasses define more geometries and allow therefore more tasks. The number of the current frame always must be in the interval `[0, number of frames - 1]` (C convention, only the frame button of the user-interface shifts the frame number).

This `torus_knot.to` file can be found in the demo directory of the **GRAPE** distribution, in the same directory there are two spline files for the `double` variables called `torus_knot_parm_0.sp` and `torus_knot_parm_1.sp`, they are loaded automatically when the `torus_knot.to` file is read in using the `read *.in` button (look at the message printed in the shell).

After reading in the file an (8, 3) torus knot should appear in the graphics window (perhaps you should also read the `torus_knot.st` status file using the `read *.st` button in the manag menu). If you change the time or the frame the torus knot radii should vary — unless the spline files were missing, in this case you first have to edit a spline to see a difference between the three frames.

There is no data for a torus knot in the description files or the splines, so how is the torus knot computed? Since the geometry of a `Time_Object` is arbitrary (in this case `Triang1d`) a `"compute-send"` method has to be provided to create the data:

```
TIME_OBJECT *time_object_compute_send()
{
  TIME_OBJECT *self;

  self = (TIME_OBJECT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /* no frames -> nothing to compute */
  if(!self->num_of_frames)
    END_METHOD(NULL);

  /* different objects are distinguished by their name (same as filename) */
  if(!strcmp(self->name, "torus_knot")) {
    TIMESTEP *step;
    TRIANG1D *triang;
    int i, m, n, num_points;
    double r, rad, torad, delta;
    int ppr = 30, ggT;

    /* copy the variables so we can use the shorter names */
    m = self->time_parm->int_parm[0];
    n = self->time_parm->int_parm[1];
    rad = self->time_parm->double_parm[0];
    torad = self->time_parm->double_parm[1];
```

```
    /*  divide out gcd to avoid identical segments */
    ggT = g_gcd(m, n);
    num_points = ppr * (m + n) / ggT;

    /* create the geometry with "alloc" and "create-objects" */
    if(!self->geometry)
      self->geometry = (CHAIN *)GRAPE(Chain, "new-instance")("geom");
    if(!self->geometry->object) {
      self->geometry->object = (INSTANCE *)
        GRAPE(TimeStep, "alloc")(self->num_of_frames, 0.0, 1.0, "geom");
      GRAPE(self->geometry->object, "create-objects")(Triang1d, "pipe");
    }

    /* allocate memory for the Triang1d instances if necessary */
    step = (TIMESTEP *)self->geometry->object;
    do {
      triang = (TRIANG1D *)step->pre_object;
      if(triang->max_number_of_points < num_points + 1)
        GRAPE(triang, "list-alloc")(num_points + 1);
      step = step->post_step;
    } while(step && step->pre_step->time < step->time);

    /* this is the frame we have to compute */
    triang = (TRIANG1D *)GRAPE(self->geometry->object, "get-pre-obj")
                             (self->time_step[self->frame]);

    /* compute the torus knot */
    delta = 2.0 * M_PI / ggT / num_points;
    r = 0.0;
    for(i = 0; i <= num_points; i++) {
      triang->x[i] = (rad + torad * cos(m * r)) * cos(n * r);
      triang->y[i] = (rad + torad * cos(m * r)) * sin(n * r);
      triang->z[i] = torad * sin(m * r);
      r += delta;
    }
    triang->number_of_points = num_points + 1;

  } else
    printf("time_object_compute_send: unknown object %s\n", self->name);

  END_METHOD(self);
}
```

If there is more than one `Time_Object` you work with the method has to find out which object should be computed, this is done by checking the `name` of the `Time_Object` instance it is called on.

For the `objects` of the `geometry Chain` the method creates a sequence of `num_of_frames` `TimeSteps` with `Triang1d` objects. This is only done once but the memory for the `Triang1d` instances is checked every time the method is called because the number of points might change. Checking only the `step` variable as exit criterion in the `while` loop isn't sufficient since the `TimeSteps` created by `"alloc"` are cyclic — the last is pointing to the first and vice versa.

The part which computes the torus knot is quite simple, but why is the torus knot time-dependent? First of all we store the computed data in the `Triang1d` instance of the step `frame`, `"compute-video-send"` computes all frames by calling `"compute-send"` for all frame numbers in `[0, num_of_frames]`. The time-dependency of the `double` variables is not that obvious, we only access the `time_parm->double_parm` values which seem to be static. But every time `"set-frame"` (and therefore `"set-time"`) is called — this is done by `"compute-video-send"` for each frame — the splines of the `double` variables are evaluated for the new time and the computed values are written to the variables!

Since curves are used quite often a special class `Time_Curve` is provided, the corresponding project is described in the following section.

8.2.2 Project *Curve*

The *Curve* project is used to compute explicitly given time-dependent curves, the class `Time_Curve` it is based on is the simplest subclass of `Time_Object`, see section 7.4.2.

The user-interface is inherited from the *Time-Object* project, the extended functionality is added by replacing the methods for reading the description files, allocating memory for the geometry and computing the data.

Additional to the *Time-Object* data for task, frame and parameter the *Curve* description files with extension `*.cv` contain information about the definition interval and the explicit formulas for the curve, see the example in section 8.2.2.2.

8.2.2.1 Using Variables in Functions

The coordinate functions and the definition interval of the curve are read as strings from an input file, compiled into an internal representation with the function compiler (see section 10.2.8) and evaluated during the computation. The function compiler is also used for other subprojects of *Time-Object* resp. subclasses of `Time_Object`, for example for the coordinate functions and the domain data of `Explicit` and the point coordinates of `Surface_Build` instances.

Besides the `int` and `double` variables provided by the superclass `Time_Object` the time and the domain variables can be used for functions:

- The time can be referenced with the variable `t`, it can be used in all functions. If you need for example a definition interval which grows linearly from `[0,` π`]` at time `t = 0` to `[0, 2`π`]` at time `t = 1` you can use

  ```
  0; (1 + t) * pi;
  ```

  The constant `pi` is predefined and can also be used in all functions.

- The domain points can be referenced in the explicit coordinate functions, for curves they are stored in the `x` variable (see the next section), for surfaces in `x` and `y` (see section 8.2.5.1 for an example).

- The variables provided by `Time_Object` are named `i`, `j`, ... and `a`, `b`, ... The first `int` variable in the file corresponds to `i`, the second to `j` and so on, the `double` variables can be referenced with `a`, `b` and so on (see also the next section for an example).

There are currently some limitations for the use of the variables, only eight variables (including the time and the domain points) are allowed for each description file and the `int` and `double` variables can for example not be used for the domain data. Using a variable which isn't defined or cannot be used will result in an error while reading the input file.

8.2.2.2 A *Time-Curve* Example

As all description files for the subprojects of *Time-Object* the *Curve* files start with the `Time_Object` part containing the task, frame and parameter data (compare the `Time_Object` example in section 8.2.1.3):

```
#
# torus_knot.cv: test object for time_curve - (m, n) torus knot
#

# task
GEOMETRY;
```

```
# frame data
0; 3;           # frame / number of frames

# parameter data
2; 2;           # number of int / double parameter

# int parameter
8;              # m -> i
3;              # n -> j

# double parameter
2.5;            # radius -> a
1.0;            # torus radius -> b

# curve parameter
331;            # number of points
0.0; 2.0 * pi;  # definition interval

# coordinate functions
(a + b * cos(i * x)) * cos(j * x);
(a + b * cos(i * x)) * sin(j * x);
b * sin(i * x);
```

The new parts are the curve parameter — an `int` value for the number of points of the curve and two functions (of `t`) describing the definition interval — and the coordinate functions (of `x`). As described in the previous section the time `t` and the `int` (`i`, ...) and `double` (`a`, ...) variables can be used for the coordinate functions.

In contrast to the torus knot *Time-Object* example this version requires no programming, but there are two thing we can't control: the number of points on the curve is fixed and not depending on the actual length of the curve as in the *Time-Object* version, and we cannot divide out the greatest common divisor of (m, n).

8.2.3 Project *Surface*

This subproject of *Time-Object* provides the basic environment for working with time-dependent surfaces, the *Build*, *Explicit* and *Amandus* projects described in sections 8.2.4–8.2.6 are based on it.

In contrast to `Time_Object` which handles arbitrary geometries the `Surface` class corresponding to the *Surface* project expects the geometries to have a specific structure with underlying `Geom2d` instances for storing the surface data, see section 7.4.3. Since the geometry structure is known most methods can be written on `Surface` and the subclasses only have to provide methods to read extended description files and to compute the data (if new instance variables are added also memory management and archiving methods are needed).

Like the `Surface` class the *Surface* project is "abstract", it extends the *Time-Object* functionality and user-interface for the subprojects but there are no description files for creating `Surface` objects. The most important features added are the handling of multiple geometries and a two-parameter family of surfaces and the reflection of whole time-dependent geometries.

8.2.3.1 Working With Several Geometries

`Surface` adds further geometries for the domain, for the gauss map (the stereographic projection of the normal map onto the plane) and for a conjugate geometry to the geometry provided by `Time_Object`. The task button `TASK:...` selects one of them as the current geometry, the pointer to it is stored in the `model` instance variable and is used by most of the `Surface` methods.

Changing the task also changes the geometry which is displayed since the current time section `"display"` shows is created from the model. To be able to view the geometries independent from the current model there is a special display method for each of them. Pressing the `geometry displays` button pops up a layer which allows to select one of the display methods.

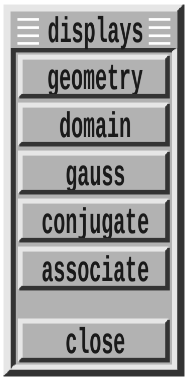


Figure 8.15: Display `Layer` for Project `Surface`

If the geometry and conjugate data set have the same structure (i.e. if the `Chains` have the same lengths and the `Geom2d` instances have the same discretizations) the special `associate` display method can be used to interpolate the current time sections of these data sets. The interpolation can be controlled by a ruler `"associate-disp"` adds to the option menu (see 7.4.3.4), together with the time ruler this allows to view a two-parameter family of surfaces.

8.2.3.2 Reflecting Time-Dependent Data

When reflecting static objects you might select three points on the object to specify a symmetry plane, or two points for a straight line, or one point for a point reflection. Then the reflection is well defined and can be executed.

For time-dependent objects this is quite different. You always see only the geometry at the current time — that means its current shape and its current discretization. A symmetry plane at another time may be different from the visible symmetry plane. So you cannot just select special points or specify a symmetry plane, you have to select a “meta structure” which is defined independent from the current time's object.

Selecting symmetry planes, lines or points is therefore done by selecting boundary curves. The curves are only referenced by the indices of the objects they belong to and their indices in the boundary curve lists of the objects, since these values are independent from the connectivity of the current time section and the position of the symmetry lines in it this allows to reflect complete time-dependent geometries.

Figure 8.16: User Interface for Project *Surface*

To understand how the reflection methods work you have to know how the data is stored. As described in section 7.4.3 the geometries of a `Surface` instance are not just single objects but each of them is a list of objects, they are stored as the `objects` of a `Chain` sequence (the index of an object is its position in this sequence). An object is reflected by adding a new `Chain` to the list, copying the object to the new `Chain`'s `object` and applying the transformation. You can either reflect a complete geometry, i.e. all objects in the list, or select which objects of the geometry should be reflected.

The `refl: config` button in the user-interface added by the *Surface* project to the interface inherited from *Time-Object* pops up a configuration layer with some checkfields which allow to configure the geometries and the objects that should be reflected (right half) and the type of transformation that should be applied (left half), see figure 8.17. Changes to the configuration are accepted when the `OK` button is pressed.

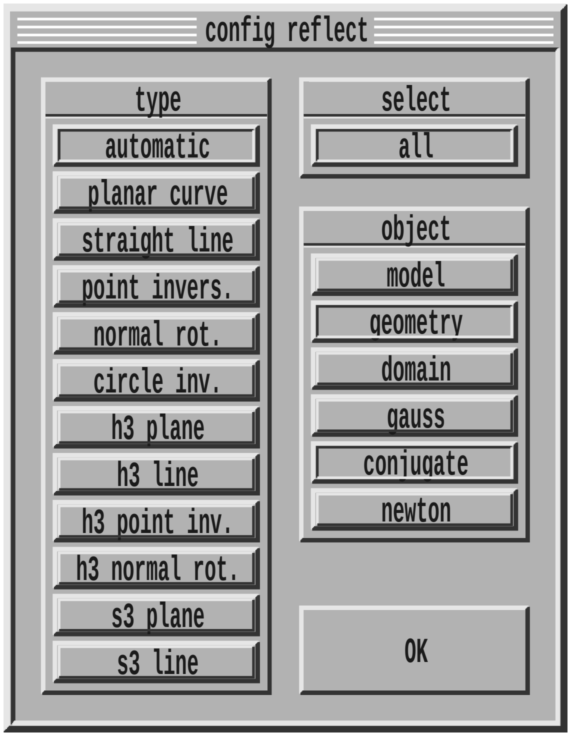


Figure 8.17: Reflection Config-Layer in Project *Surface*

Only one reflection type can be active at a time, the default is `automatic`. In this mode the program detects automatically whether a boundary curve is planar or straight and applies a plane reflection or 180 degree rotation. The reflection types provided are reflection at a plane, 180 degree rotation around a straight line (you only have to use these two when the automatic detection fails), inversion at a point, 180 degree rotation around the normal of a point and inversion at a circle. Some of them are also available for hyperbolical and spherical geometries.

If the `all` checkbox is on (default) the reflection is applied to all objects of the chosen geometries, otherwise you can select the objects you want to reflect with the mouse. The remaining checkboxes allow to select the geometries that should be reflected, by default only the `model` checkbox is on.

Since the model is displayed during the selection of the boundary curves you should be careful when other geometries are chosen for the reflection. Actually this is only useful if the `geometry` and the `conjugate` geometries are really conjugate surfaces and you want to reflect both at the same time, for example for viewing the associate surface. In this case either `geometry` or `conjugate` should be chosen as model, applying a plane reflection to the model then results in an 180 degree rotation of the other geometry and vice versa.

Pressing the `reflect` button changes the display to reflection mode. Only mouse clicks in the graphics window are accepted in this mode, the left mouse button is used to pick boundary curves or objects by clicking on points on them, the right mouse buttons accepts the choices or executes the reflection and the middle mouse button exists without changes. If the graphicdevice is in grid mode (this is strongly recommended) the model will be drawn in green, selected boundary curves or objects in red.

For most reflection types only one boundary curve has to be selected, but for types like normal rotation and point inversion a point is needed. In this case you have to select two curves with a common end point, this point then is used for the reflection. If the `all` checkbox is off you also have to select the objects you want to reflect before you can finally execute the reflection.

With the `refl: del last` button you can undo reflections. Using this button for the first time after some reflections will delete all objects created by the last reflection, subsequent uses will delete only the last object of the geometry, i.e. you can delete the objects created by the previous reflections only one by one.

8.2.3.3 Handling the `Geom2d` Instances

The surface data of the geometries is stored in `Geom2d` instances therefore it is possible to select whether the surface, its boundary curves or parameter lines should be displayed (by setting or deleting the correct bits in the `Geom2d` flags, compare 6.3.4). This is what the second row of the *Surface* user-interface shown in figure 8.16 is used for:

`geom`	This checkbox selects if the `Geom2d` surfaces should be shown. Each geometry has its own instances with their own flags therefore the selection has to be done for each geometry separately.
`bnd`	Selects whether the boundary should be drawn. Some rulers will be added if this checkbox is on, they allow to control the discretization and the radius of the tube drawn around the boundary lines.
`hor`, `ver`	Same for the horizontal and vertical parameter lines. The lines might not be available, for example when they were not computed for the original object before it was reflected and merged or closed.

8.2.4 Project *Build*

With this subproject of *Surface* it is possible to generate topological surfaces of arbitrary type and genus. The surfaces are created by pasting triangular or quadrilateral patches together, the description of the patches is read from an input file with extension `*.bd`.

The *Build* project is based on class `Surface_Build` (see 7.4.4) which provides the methods to read the description files and to compute the patches from the data.

This project together with the *Refine* project is especially useful for creating initial surfaces for the minimization algorithms of the *Dual Project* described in section 8.2.7.

8.2.4.1 The *Build* Input File Format

Like the *Curve* description files the *Build* files start with the *Time-Object* part containing the task, frame and parameter data. Additionally the *Build* files contain the data for the points the surface patches are build from and for the patches themselves. The format used to describe the points and patches is quite complex therefore we first explain it before presenting the complete examples in the next section.

Points are defined by three coordinate functions (the time `t` and the `int` and `double` parameter can be used for them, see 8.2.2.1) and three `int` flags, one for each function:

```
xflag; yflag; zflag;   xfunction; yfunction; zfunction;
```

If a flag is `0` nothing special happens, when the function is read it is compiled with the function compiler (see 10.2.8) and later evaluated during the computation of the surface. But if some other value is given a ruler is added to the option menu and the corresponding function is used to create an initial spline for it, this way you can edit the point interactively.

The interesting point is that coordinates with identical non–zero flags are connected to the same ruler, therefore they can be edited simultaneously. As an example we use the points of the CLP example in the next section:

```
# ruler flags / point coordinates
0; 0; 0;       0.0;     0.0;     0.0;
1; 0; 0;   0.5 + t;     0.0;     0.0;
1; 2; 0;   0.5 + t; 0.5 + t;     0.0;
0; 2; 0;       0.0; 0.5 + t;     0.0;
0; 0; 3;       0.0;     0.0; 1.5 - t;
1; 0; 3;   0.5 + t;     0.0; 1.5 - t;
```

Each line contains the data for one of the six points, only the time `t` is used in this example. Three rulers will be created, one for the x–coordinates of points 1, 2 and 5, one for the y–coordinates of points 2 and 3 and one for the z–coordinates of points 4 and 5 (points are numbered from 0 as arrays in C).

The function corresponding to a non–zero flag is only evaluated when the flag value is encountered for the first time, otherwise it is just read as a string and discarded. Therefore we could also use the line

```
1; 2; 0;   ruler; 0.5 + t; 0.0;
```

for point 2 of the example since the ruler for the flag value 1 was already created from the x–coordinate of point 1.

Patches are defined by their vertices, the boundary curves connecting them and some triangulation information. There are many options, but if you only need straight boundary lines connecting the vertices the format is simple:

```
orient; 0; type;   v0; v1; v2; [v3;]   ulines; vlines; usub; vsub;
```

The values `v0` upto `v3` are the indices of the vertices in the point list, the `type` can be either 4 for a quadrilateral patch or 3 for a triangular patch (in this case `v3` is missing). `ulines` is the number of points in the direction of the boundary between the points `v0` and `v1`, `usub` is the number of additional points between each two neighbours of the `ulines` points. The value `vlines` and `vsub` are only evaluated for quadrilateral patches (but they always have to be present), they define the numbers of points in the other direction.

The method `"assign-cons-send"` computes parameter lines only for the `ulines/vlines` lines but not for those added by `usub/vsub`, this allows to define a grid with parameter lines on a subgrid. This is more interesting for `Explicit` and `Amandus` instances where parameter lines make sense, for `Surface_Build` instances they would be destroyed by the refine methods and especially by `"mollify-send"` and `"equiangulate-send"` (compare 8.2.7) therefore by default they are not computed.

For the CLP example we have two quadrilateral patches (0, 1, 2, 3) and (1, 0, 4, 5) with a basic triangulation, the resulting surface family is shown in figure 8.18:

```
# flag, bnd type, patch type / point indices / u- and v-lines, sub-points
0; 0; 4;   0; 1; 2; 3;   7; 7; 0; 0;
0; 0; 4;   1; 0; 4; 5;   7; 7; 0; 0;
```

For common boundary lines of the patches — in this case the line from point 0 to point 1 — the number of points have to be identical otherwise it is not possible to merge and close the patches to create a single surface.

The orientation of the triangles of a patch and therefore the direction of the surface normal is induced by the orientation of the patch, i.e. the order of its vertices. The orientation of a patch can be changed either by setting the `orient` flag to a non–zero value or by reversing the order of the points.

Merging and especially closing patches with different orientations can cause severe problems or even crashes since some methods rely on the correct orientation of the triangles. Therefore the common boundary line of two patches should have different (global) orientations for both patches — as the common line in the CLP example which runs from point 0 to point 1 in the first patch and in the other direction in the second patch. If you are not sure whether the patches of you surface have the correct orientation you should use the display method `"normals-disp"` to check the surface normals.

Sometimes it is impossible to orientate all patches in the same way (for example for non–orientable surfaces or if more than two patches meet in a common line), in this case you have to be very careful which methods you use. `"close-send"` and the minimization algorithms of the *Dual* project (see 8.2.7) are able to handle these surfaces, but methods like `"equiangulate-send"`, `"mollify-send"` and the refine methods which rely on the `neighbour` information of the `Geom2d` instances will get you into trouble.

Additional to straight lines other types of curves can be used to connect the vertices of a patch, in this case the zero between `orient` and `type` has to be replaced by a non–zero value and for each boundary the type has to be specified:

```
orient; 1; type;   v0; v1; v2; [v3;]   ulines; vlines; usub; vsub;
   type0; 0; [ind00; ind01;]
   type1; 0; [ind10; ind11;]
   type2; 0; [ind20; ind21;]
  [type3; 0; [ind30; ind31;]]
```

The first additional line describes the curve from `v0` to `v1`, the second the one from `v1` to `v2` and so on, in case of a triangular patch the last line is missing (as `v3`). The curve types have to specified as octal values:

`00200` – The two vertices are connected with a straight line. The value following the type is currently not used and must be zero for all types, for straight lines the two values `ind?0` and `ind?1` are missing.

`00400` – A circle is used to connect the vertices. Like for the following types the two integer values `ind?0` and `ind?1` refer to the point list, in this case `ind?0` is the index of the midpoint of the circle and `ind?1` the one of the normal of the circle plane. As usual the variables can be used to define the midpoint and the normal, and their coordinates can be bound to rulers to edit them interactively.

`01000` – Connect the vertices with a helix. `ind?0` and `ind?1` are the two points on the axis of the helix with the same heights (with respect to the axis) as the vertices, i.e. `ind?0` must be in the plane orthogonal to the axis through the start vertex of the helix and `ind?1` in that one through the end vertex with the same normal.

`02000` – Use a spline to connect the vertices. The `ind?0` and `ind?1` values refer to the tangent vectors at the start and the end vertex of the spline.

For some applications, for example for the minimization algorithms of the *Dual* project (see 8.2.7) and the refine methods, it is necessary to define restrictions for the boundary curves. This is what the lower bits of the type field are reserved for:

`00004` – Restrict the points of the curve to a circle (of course this value can only be used for circular curves of type `00400`), currently this value is only used by the refine methods.

`00010` – All points of the boundary curve are fixed. This is the default for most methods if no restriction is given, but some methods detect automatically whether a curve is straight or planar if no restriction is given therefore this value should be set if you want to make sure that the points of a straight or planar curve remain fixed.

`00020` – The points of a boundary curve are restricted to a straight line, this is only possible for straight curves with type `00200`. This value isn't really needed at the moment (new points inserted by refine are already on the curve and the minimization methods detect straight lines) but it should be set anyway.

`00040` – Restrict the points of the boundary curve to a plane. Currently the plane is computed from the curve data, therefore this value can only be used for circular curves of type `00400` and for spline curves of type `02000` if they are planar.

Some example `*.bd` input files which use these curve types can be found in the demo directory of the **GRAPE** distribution, one of them is presented in the next section.

8.2.4.2 *Surface-Build* Examples

The first example was already discussed in the previous section, it creates the conjugate boundary contour and an initial triangulation for a family of triply periodic minimal surfaces including the CLP surface.

Like for all other subprojects of *Time-Object* the description file `CLP.bd` is read in with the `read *.in` button, for `Surface_Build` instances the surfaces then are computed automatically. The `CLP.bd` description file should be located in the `geometry/build_diri` subdirectory of the **GRAPE** demo directory:

```
#
# CLP.bd: test object for surface_build - one-parameter family of CLP surfaces
#

# task
GEOMETRY;


# frame data
0; 5;           # frame / number of frames


# no parameter
0; 0;


# number of parts / number of points
2; 6;


# ruler flags / point coordinates
0; 0; 0;        0.0;     0.0;     0.0;
1; 0; 0;    0.5 + t;     0.0;     0.0;
1; 2; 0;    0.5 + t; 0.5 + t;     0.0;
0; 2; 0;        0.0; 0.5 + t;     0.0;
0; 0; 3;        0.0;     0.0; 1.5 - t;
1; 0; 3;    0.5 + t;     0.0; 1.5 - t;


# flag / type / patch type / point indices / u- and v-lines, sub-points
0; 0; 4;    0; 1; 2; 3;    7; 7; 0; 0;
0; 0; 4;    1; 0; 4; 5;    7; 7; 0; 0;
```

The second example is just a quadrilateral patch with two straight lines, one circle arc and a spline as boundaries, the corresponding file is `dummy.bd`. In this example not the points but the spline tangents can be edited interactively since they are bound to rulers:

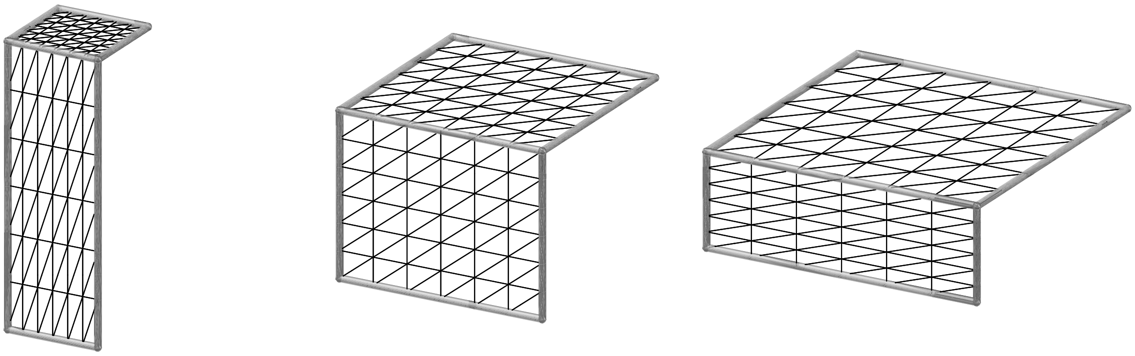


Figure 8.18: Conjugate boundary contour for a family of triply periodic minimal surfaces with a basic triangulation. The picture shows three out of the five frames created by reading the `CLP.bd` description file including the conjugate contour of the CLP surface in the middle.

```
#
# test quadrilateral with circle and spline boundaries
#

# task
GEOMETRY;


# frame data
0; 1;          # frame / number of frames


# no parameter
0; 0;


# number of patches / number of vertices
1; 8;


# ruler flags / point coordinates
0; 0; 0;   0.0; 0.0; 0.0;
0; 0; 0;   1.0; 0.0; 0.0;
0; 0; 0;   1.0; 1.0; 0.0;
0; 0; 0;   0.0; 1.0; 0.0;
0; 0; 0;   0.5; 1.0; 0.0;    # circle midpoint
0; 0; 0;   0.0;-1.0; 0.0;    # circle normal
1; 2; 1;   0.2; 1.0; 0.2;    # spline tangent at start point
3; 0; 4;   1.0; 0.0;-3.0;    # spline tangent at end point


# flag / type / patch type /point indices /u- and v-lines, sub-points
0; 1; 4;  0; 1; 2; 3;   7; 4; 0; 0;
   02010; 0; 6; 7;           # spline with fixed points
   00220; 0;                 # straight line
   00404; 0; 4; 5;           # circle with circle restriction
   00220; 0;                 # straight line
```

Because the spline is not planar the only option is to fix its points, for the straight lines the points are allowed to move along them. For the circle we could use the restrictions `00004` (restrict the points to the circle), `00040` (restrict the points to the plane of the circle) and of course `00010` (fully fixed). Add the *Diri* project (which includes *Build*), read in the file and try out `"mollify-send"` with the boundary option enabled (see 7.4.3.7) or the refine methods (see 8.2.8) to see how the circle restriction works.

8.2.5 Project *Explicit*

With the *Explicit* project you can compute explicitly given (i.e. parametrized) time-dependent surfaces. The corresponding `Explicit` class provides methods to manage the domain and to read the domain data and the coordinate functions from description files.

Since *Explicit* is a subproject of *Surface* it inherits the user-interface described in sections 8.2.1.1 and 8.2.3.2, the functionality of the *Surface* project is extended by replacing the methods for reading the description files and for computing the data.

Usually no programming is required for using this project, but if you want to compute surfaces which for example need a special domain or parameter handling or cannot be described with the formulas in the description file you can provide you own domain transformations or replace the `"compute-explicit"` method (see sections 8.2.5.2 and 8.2.5.3).

8.2.5.1 An `Explicit` Example

Additional to the *Time-Object* data for task, frame and parameter the *Explicit* description files with extension `*.ex` contain information about the domain of the surface and the explicit formulas for the surface.

By default a rectangular domain `[umin, umax]` $\times$ `[vmin, vmax]` with a basic grid of `ulines` points in the u–direction and `vlines` points in the v–direction is used for *Explicit* surfaces:

```
# domain triangulation: u- and v-lines, sub-points
ulines; vlines; usub; vsub;

# domain size: umin / vmin / umax / vmax
umin; vmin; umax; vmax;
```

The domain is time-dependent, i.e. the variable `t` can be used for `umin`, `vmin`, `umax` and `vmax`.

The sub-points allow to refine the grid of the `ulines` $\times$ `vlines` points. `usub/vsub` additional points are inserted between each two neighbours of the `ulines/vlines` points, therefore the total number of points in the u–direction is `ulines` + (`ulines` − 1) ∗ `usub`.

Of course you can also create a finer grid without using `usub/vsub` by just increasing `ulines/vlines`. Using sub-points only makes sense in connection with parameter lines which are computed by `"assign-cons-send"` (see 7.4.5.6). The parameter lines are only computed for the basic `ulines` $\times$ `vlines` grid therefore distinguishing between points and sub-points makes it possible to compute the surface on a finer grid than the one used for the parameter lines.

After reading in a description file you will find in the option menu additional to the parameter and domain min/max rulers two rulers controlling the distribution of the points in u– and v–direction. These point distributions can't be defined in the description file but of course they can be edited interactively and then saved, the next time the description file is read the saved splines are found and read automatically.

With the following example description file you can compute the logo of the SFB 256, the Henneberg surface, which can be found on all publications of the SFB, for example on the title page of the printed **GRAPE** manual and on the **GRAPE** home page http://www-sfb256.iam.uni-bonn.de/grape/main.html:

```
#
# GRAPE Logo: The Henneberg Surface
#
```

```
# task
GEOMETRY;

# frame / number of frames
0; 1;

# no parameter
0; 0;

# part / number of parts
1; 1;

# flag (unused)
0;

# domain triangulation: u- and v-lines, sub-points
11; 80; 0; 0;

# domain size: umin / vmin / umax / vmax
pi/5.0; 0.0; 2.0*pi/5.0; 2.0*pi;

# coordinate functions
-1.0 + cosh(2.0 * x) * cos(2.0 * y);
-sinh(x) * sin(y) - sinh(3.0 * x) * sin(3.0 * y) / 3.0;
-sinh(x) * cos(y) + sinh(3.0 * x) * cos(3.0 * y) / 3.0;
```

Since currently only one part (surface patch) is supported the part numbers always have to be set to one, the flag is not yet used and should be set to zero.

For the three coordinate functions the (time-dependent) `double` variables and the time `t` can be used (refer to section 8.2.2.1 for naming conventions), the points in the domain are referenced with (`x`, `y`).

8.2.5.2 User–Defined Domain Transformations

For class `Time_Object` the user must replace the `"compute-send"` method to compute his own objects (see section 8.2.1, especially 8.2.1.3), but for `Explicit` this is different. Since the geometry structure for `Surface` instances is fixed — both the `domain` and the `geometry` consist of a list of `TimeStep` sequences with `Geom2d` instances — the `Explicit` `"compute-send"` method can handle all the memory management and compute the data. The computation is done in three steps:

- The rectangular default domain `[umin, umax]` × `[vmin, vmax]` with a basic grid of `ulines` × `vlines` points and `usub`/`vsub` sub-points is computed (see the previous section).
- If present the user–defined domain transformation is applied to the domain points (see below), the resulting points are stored in the `domain` geometry of the `Explicit` instance, if no domain transformation is given the default points are stored instead.
- The `"compute-explicit"` method is called for all domain points to compute the corresponding surface points. The `"compute-explicit"` method can be replaced by the user if necessary (see the next section).

A domain transformation can be defined by setting the `compute_arg` pointer of the `Explicit` instance (see 7.4.5.1), this is usually done by replacing the (empty default) `"set-compute-arg"` method which is called when a description file is read in:

```
EXPLICIT *explicit_set_compute_arg()
{
  EXPLICIT *self = (EXPLICIT *)START_METHOD(G_INSTANCE);
```

```
  ASSURE(self, "", END_METHOD(NULL));

  self->compute_arg = transform_domain;

  END_METHOD(self);
}
```

The name of the transformation function is arbitrary, its arguments are the `Explicit` instance (to be able to access the `Explicit` name and parameter), the number of the surface part that is computed (only used by subclasses at the moment), and the coordinates of the point that should be transformed. The transformed point has to be returned in a `COMPLEX` variable:

```
static COMPLEX transform_domain(EXPLICIT *explicit, int part, double u, double v)
{
  if(!strcmp(explicit->name, "paraboloid")) {
    double a = explicit->time_parm->double_parm[0];
    double b = explicit->time_parm->double_parm[1];
    return g_cmplx(a * (u + v) / 2.0, b * (u - v) / 2.0);
  }

  return g_cmplx(u, v);
}
```

As `Time_Object "compute-send"` (compare 8.2.1.3) the domain transformation function has to check the name of the instance to find out which surface should be computed. Since the transformation function is called for all instances it also has to handle the default case, i.e. return `g_cmplx(u, v)` if nothing special has to be computed.

This example transformation function belongs to an input file `paraboloid.ex` with two `double` parameter which computes a hyperbolical paraboloid:

```
# Hyperbolical Paraboloid

GEOMETRY;         # task
0; 1;             # frame / number of frames

0; 2;             # number of int / double parameter

# double parameter
1.0;              # a
2.0;              # b

1; 1;             # part number / max number of parts

# part 1
0; 10; 10; 0; 0; # flag / number of u- and v-lines / u- and v-sub
-1; -1; 1; 1;     # umin / vmin / umax / vmax

# formulas
x;
y;
x^2 / a^2 - y^2 / b^2; # + for elliptical paraboloid
```

The description file also can be used without the transformation function but with the function the parameter lines (which can be computed by sending `"assign-cons-send"` to the `Explicit` instance and switched on with the $\boxed{\texttt{hor}}$ and $\boxed{\texttt{ver}}$ checkboxes) are more interesting, they are straight lines on the surface.

Of course we really don't need the transformation function for this example, we just have to substitute $\mathtt{a} * (\mathtt{x} + \mathtt{y})/2$ and $\mathtt{b} * (\mathtt{x} - \mathtt{y})/2$ for `x` and `y` in the formulas of the description file (i.e. use $\mathtt{a} * (\mathtt{x} + \mathtt{y})/2$; $\mathtt{b} * (\mathtt{x} - \mathtt{y})/2$; $\mathtt{x} * \mathtt{y}$; as formulas) to get the same result.

Domain transformations for subclasses of **Explicit** are implemented in the same way, of course it is possible to have different transformation functions for subclasses by providing "set-compute-arg" methods for them. In section 8.2.6.1 you can find an example for the **Amandus** subclass of **Explicit**.

8.2.5.3 Replacing the "compute-explicit" Method

You can replace this method with your own **"compute-explicit"** method if you want to compute something special that cannot be described with the input file, for example if you have to compute some parameter for the surface numerically, if you need some function not supported by the function compiler (see 10.2.8 for a list of the supported functions) or if you want to use the same parameter for several surfaces.

The first argument of the **"compute-explicit"** method is the **Geom2d** instance the data is stored in, the second is the index of the point that should be computed and the third one are the coordinates of the domain point. As an example we use the stereographic projection of helicoids in the three-sphere:

```
EXPLICIT *explicit_compute_explicit(GEOM2D *geom, int num, COMPLEX z)
{
  EXPLICIT *self;
  double arg[10], x, y;
  int i;

  self = (EXPLICIT *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /* copy the double parameter to the arg array (needed for fn_eval below) */
  x = arg[0] = z.x;
  y = arg[1] = z.y;
  for(i = 0; i < self->time_parm->num_double_parm; i++)
    arg[i + 2] = self->time_parm->double_parm[i];

  if(!strcmp(self->name, "s3helicoid")) {
    extern int proj_stereo(VEC3, VEC4);
    double alpha = self->time_parm->double_parm[0];
    VEC4 x4;
    VEC3 x3;

    x4[1] = cos(y) * cos(x);
    x4[2] = cos(y) * sin(x);
    x4[3] = sin(y) * cos(alpha * x);
    x4[0] = sin(y) * sin(alpha * x);

    proj_stereo(x3, x4);

    geom->x[num] = x3[0];
    geom->y[num] = x3[1];
    geom->z[num] = x3[2];
  } else {
    /* the default case */
    geom->x[num] = fn_eval(self->coord_f.x, arg);
    geom->y[num] = fn_eval(self->coord_f.y, arg);
    geom->z[num] = fn_eval(self->coord_f.z, arg);
  }

  END_METHOD(self);
}
```

The part upto the `if` and from the `else` on always has to be present to handle the default case, i.e. to evaluate the formulas read from other description files. Again the `name` of the `Explicit` instance is used to determine which surface should be computed.

The description file `s3helicoid.ex` only contains the parameter and domain data, the formulas have to be present but they are not used by `"compute-explicit"`:

```
# S^3 Helicoid

GEOMETRY;         # task
0; 1;             # frame / number of frames

0; 1;             # number of int / double parameter
0.0;              # double parameter a

1; 1;             # part number / max number of parts

# part 1
0; 20; 20; 0; 0; # flag / number of u- and v-lines / u- and v-sub
0; 0; pi; pi;    # umin / vmin / umax / vmax

# dummy formulas
0; 0; 0;
```

Of course we can avoid writing a new `"compute-explicit"` method for this example by coding the stereographic projection into the explicit formulas in the description file:

```
cos(y) * cos(x) / (1 + sin(y) * sin(a * x));
cos(y) * sin(x) / (1 + sin(y) * sin(a * x));
sin(y) * cos(a * x) / (1 + sin(y) * sin(a * x));
```

8.2.6 Project *Amandus*

The *Amandus* subproject of *Explicit* is used for working with minimal surfaces defined via the Weierstrass representation formula (another project dealing with (approximations of discrete) minimal surfaces is the *Dual* project, see section 8.2.7). The corresponding class `Amandus` adds instances variables for the Weierstrass data and an additional geometry (which is needed for the analytic continuation of functions) to its superclass `Explicit`, see section 7.4.6.1.

There is a large number of *Amandus* surfaces build into **GRAPE**, the corresponding description files should be located in the `geometry/amandus` subdirectory of the demo directory. You should also take a look at our **GRAPE** Minimal Surface Library at http://www.iam.uni-bonn.de/sfb256/grape/EXAMPLES/AMANDUS/amandus.html.

Implementing a new *Amandus* surface always requires programming, the Weierstrass functions and a function for computing the domain of the surface have to be linked to the **GRAPE** executable, see section 8.2.6.1.

Two different versions of the Weierstrass representation formula are available, the local version

$$\int (1-g^2,\, i(1+g^2),\, 2g)\, f\, dz$$

where only the two complex functions `g(z)` and `f(z)` have to specified and the global version

$$\int \left(g-\frac{1}{g},\, i(g+\frac{1}{g}),\, 2\right) dh$$

where the differential `dh` is implicitly given by the function `poly(g, dh)`. The derivative of this function with respect to `g` has to be supplied in `dzpoly`, supplying the derivative `dwpoly` of `poly` with respect to `dh` is optional but greatly speeds up the Newton method.

For a description of the construction of minimal surfaces via the Weierstrass representation formula refer to [Kar1] or [Kar2] where you can also find the Weierstrass data for most of the examples of the **GRAPE** Minimal Surface Library.

8.2.6.1 An *Amandus* Example

For simplicity we only describe the implementation of an *Amandus* surface using the local Weierstrass representation formula (if you want to use the global version and need help please send a mail to grape@iam.uni-bonn.de). As an example we use the Riemann Step surface, a plane which is punctured at $\pm\rho$ to grow one upward and one downward catenoid end. See [Kar2] and http://www.iam.uni-bonn.de/sfb256/grape/EXAMPLES/AMANDUS/amandus.html for more information and pictures.

The *Amandus* description files with extension `*.am` start with the *Time-Object* data for task, frame and parameter, followed by the domain data of the surface. The domain data format is identical to that one used for *Explicit* surfaces (see section 8.2.5.1), but for *Amandus* surfaces more than one domain part is allowed. Finally the type of the Weierstrass representation formula to be used has to be specified, `KARL` (Weierstrass) for the local and `HERMANN` (Amandus Schwarz) for the global version. The description file for the Riemann Step example is called `riem_step.am`:

```
#
# Riemann Step Surface: plane with two catenoid ends
#

GEOMETRY;       # task
0; 1;           # frame / number of frames

0; 2;           # number of int / double parameter

# double parameter
1.0;            # rho
0.8555996772;   # r = (-rho^2 + (4 rho^2 - 1)^(1/2))^(1/2) for closed period

2; 2;           # part / number of parts

# part 1
0; 8; 10; 0; 1; # flag / number of u- and v-lines / u- and v-sub
3; 0; 0; pi;    # umin / vmin / umax / vmax

# part 2
0; 6; 10; 0; 1; # flag / number of u- and v-lines / u- and v-sub
0; 0; -2; pi;   # umin / vmin / umax / vmax

KARL;           # type: local Weierstrass representation
```

The (rectangular) domain parts of an *Amandus* surface are transformed in the same way as the domain of an *Explicit* surface, the `"set-compute-arg"` method which sets a `compute_arg` function has to be replaced (a detailed description can be found in section 8.2.5.2). In this example we have two domain parts which are mapped to a quarter sphere but circles around ρ (catenoid end) and ∞ (planar end), see figure 8.19:

```
static COMPLEX transform_domain(AMANDUS *amandus, int part, double u, double v)
{
  if(!strcmp(amandus->name, "riem_step")) {
    double rho = amandus->time_parm->double_parm[0];

    switch(np) {
    case 0:
      return g_crmul(g_csqrt(g_cadd(g_cexp(g_cmplx(u / rho, v)), g_CE)), rho);
      break;
    case 1:
```

```
      return g_crmul(g_csqrt(g_cadd(g_cexp(g_cmplx(u * rho, v)), g_CE)), rho);
      break;
    }
  }

  return g_cmplx(u, v);
}


AMANDUS *amandus_set_compute_arg()
{
  AMANDUS *self;

  self = (AMANDUS *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  self->compute_arg = transform_domain;

  END_METHOD(self);
}
```

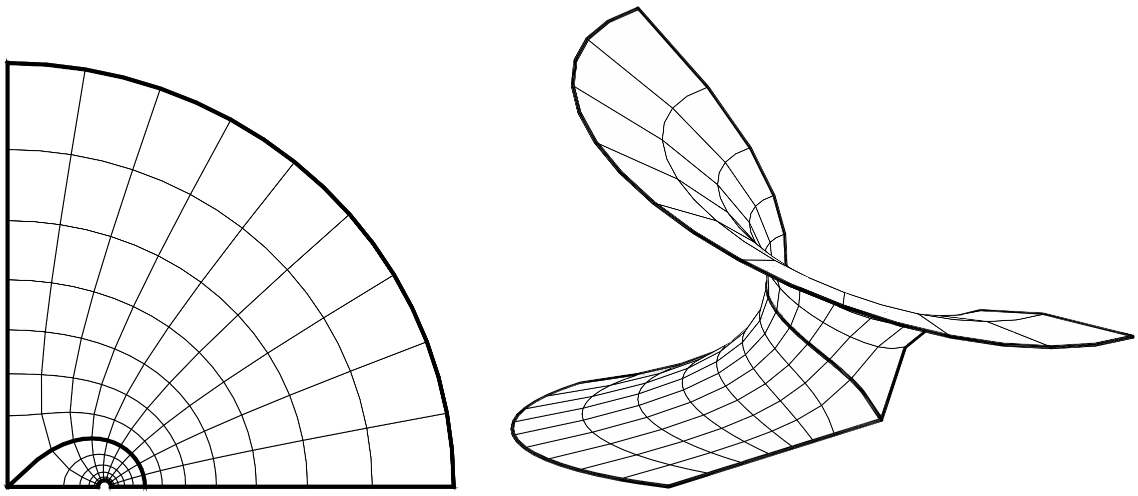


Figure 8.19: Fundamental domain of the Riemann Step surface (right picture) computed by integration of the Weierstrass data along the parameter lines in the domain (left picture). The straight boundary lines of the domain are mapped to straight/planar boundary lines of the surface, reflection of the fundamental piece on these lines gives the complete surface.

For the local version of the Weierstrass representation formula we have to define the two complex functions `g(z)` (the Gauß map) and `f(z)`, the differential `dg` of `g` is not needed for computing the surface but there are other methods like `"gausscurv-disp"` which use it therefore we define it:

```
/*
 * riem_step
 *
 * g  = z^2 - r^2
 * f  = - rho^2 / r^2 / (z^2 - rho^2)^2
 * dg = 2 z
 */

static double *riem_step_rho, *riem_step_r;
```

```
static COMPLEX riem_step_g(COMPLEX z)
{
  return g_csub(g_cmul(z, z), g_cmplx(*riem_step_r * *riem_step_r, 0.0));
}

static COMPLEX riem_step_f(COMPLEX z)
{
  double rhosqr = *riem_step_rho * *riem_step_rho;

  return g_crmul(g_cpower(g_csub(g_cmul(z, z), g_cmplx(rhosqr, 0.0)), -2),
                 -rhosqr / (*riem_step_r * *riem_step_r));
}

static COMPLEX riem_step_dg(COMPLEX z)
{
  return g_crmul(z, 2.0);
}


AMANDUS *amandus_set_weier_f()
{
  AMANDUS *self;

  self = (AMANDUS *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  if(!strcmp(self->name, "riem_step")) {
    riem_step_rho = &self->time_parm->double_parm[0];
    riem_step_r   = &self->time_parm->double_parm[1];

    self->weier_f.g  = riem_step_g;
    self->weier_f.dh = riem_step_f;
    self->weier_f.dg = riem_step_dg;
  } else
    fprintf(stderr, "set-weier-f: surface \"%s\" unknown\n", self->name);

  END_METHOD(self);
}
```

The Weierstrass functions are added by replacing the empty default `"set-weier-f"` method, again the `name` of the `Amandus` instance is used to determine which functions should be used to initialize `weier_f`. If the (time-dependent) parameter are needed for the Weierstrass functions they have to be copied to local variables in `"set-weier-f"` since they cannot be accessed directly in the `weier_f` functions.

8.2.7 Project *Dual*

The *Dual* project allows to compute approximations of discrete minimal surfaces and their conjugates, it implements the algorithms described in [PP] (see the next section for a short overview). Additionally the *Dual* project provides methods for creating initial triangulations for the minimization algorithm, see sections 8.2.7.3 and 8.2.9.

8.2.7.1 The Minimization and Conjugation Algorithms

The algorithms are based on a discrete model, instead of approximating the continuous case discrete surfaces (i.e. triangulations) are considered and the terms from the continuous case are redefined on the discrete level.

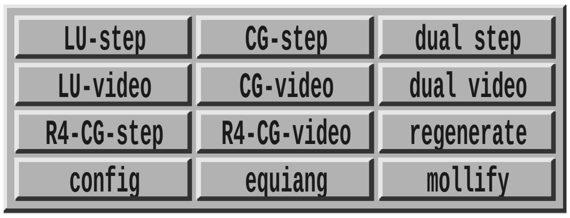


Figure 8.20: User Interface of Project *Dual*

A map between discrete surfaces with the same underlying topological simplicial complex (i.e. triangulations with the same discretization) maps corresponding points onto each other, it is linear on each triangle. The Dirichlet energy $E_D(f)$ of such a map is defined as the sum of the Dirichlet energies of the (smooth) linear maps of the triangles. It turn out that there is an explicit representation of $E_D(f)$ in terms of the angles of the domain triangles and the side lengths of the image triangles:

$$E_D(f) = \frac{1}{4} \sum_{\text{edges } i} (\cot \alpha_i + \cot \beta_i)|a_i|^2$$

where a_i are the edges in the image and α_i and β_i are the angles in the domain opposite to the edge corresponding to a_i.

Analogous to the continuous case a discrete harmonic map is defined as a critical point of the Dirichlet energy functional with respect to variations of the interior image vertices, to be able to model free boundary problems boundary points on straight resp. planar symmetry lines are allowed to vary along the image line resp. within the image plane. The condition for f to be a critial point is that for each (interior) image point p

$$\frac{1}{2} \sum_{\substack{\text{vertices } q_i \\ \text{adjacent to } p}} (\cot \alpha_i + \cot \beta_i)(p - q_i) = 0.$$

The Dirichlet energy can be written as a quadratic form $E_D(f) = P^t S P$ with a suitable representation P of the image points and a symmetric positive definite sparse matrix S. For computing a discrete harmonic map this quadratic form has to be minimized, two different methods are available:

$\boxed{\texttt{LU-step}}$, $\boxed{\texttt{LU-video}}$ Solve the problem directly by computing an LU decomposition of the matrix S. If this method is used the conjugation algorithm (see below) cannot be applied to the solution because all boundary points are assumed to be fixed. Another drawback is that the full matrix has to be stored therefore this method should only be used for small numbers of points (for larger numbers of points this method is much slower then the CG method anyway).

$\boxed{\texttt{CG-step}}$, $\boxed{\texttt{CG-video}}$ Use a conjugate gradient method for sparse matrices to compute the minimum. This method also handles boundary points on symmetry lines, it must be used to be able to apply the conjugation algorithm.

Both the LU and the CG method can be applied to a single geometry of class **`Geom2d`** or one step of a **`Surface`** instance using the $\boxed{\texttt{-step}}$ buttons. The $\boxed{\texttt{-video}}$ methods can only be used for a **`Surface`** instance, in this case the minimization method is applied to all steps. The methods on **`Surface`** copy the domain surface (which is needed for the conjugation) to the **`domain`** geometry before applying the minimization algorithms.

There is also a version of the Dirichlet minimizer for surfaces in spherical and hyperbolic space (the $\boxed{\texttt{R4-CG-}}$ methods) but it's quite difficult to use it because the metrics and projections (especially in the hyperbolic case) have to be handled.

To compute a discrete minimal surface the computation of discrete harmonic maps has to be iterated. Starting with an initial surface M_0 $(i = 0)$ the surface M_{i+1} is computed from the given surface M_i as the minimum of $F_i(X) := E_D(f : M_i \to X)$ where X has the same discretization and boundary restrictions as M_i. It can be shown that if no triangles degenerate (a subsequence of) the surfaces converges uniformly to a limit surface M, a discrete minimal surface.

The conjugation algorithm is defined for discrete harmonic maps therefore it can not only be applied to the limit surface M but also to all intermediate surfaces M_i — the minimization algorithm only has to be applied once to be able to compute a conjugate surface. Since the algorithm uses the discrete data from the minimization process directly to compute the discrete conjugate surface it avoids the problem that occur when the smooth case is simulated.

This is how the discrete conjugate surface is computed (please see [PP] on how boundary points are handled and why the result can be called the conjugate surface). The minimality condition geometrically means that for each point p the weighted edges emanating from p add up to zero, therefore they can be arranged to a closed polygon, the dual cell of p. The dual cells can be assembled to a well defined dual graph and the centers of the cells give the discrete conjugate surface which has the same discretization as the surfaces M_i (both the dual cells and the conjugate surface are drawn while they are computed).

The algorithm can be applied to the current step/all steps of a `Surface` instance with `dual step`/`dual video`, the result is stored in the `conjugate` geometry which can be viewed by setting the display method to `"conjugate-disp"` or switching the task to `TASK: CONJUGATE` (see 8.2.3.1). The LU minimization can only be used to compute the discrete harmonic map if all boundary points are fixed, if there are boundary points on symmetry lines the CG minimization algorithm has to be used to be a able to conjugate the surface.

8.2.7.2 Configuration Options

Pressing the `config` button pops up the configuration layer (figure 8.21) for the Dirichlet minimizers, it contains some rulers and checkboxes which allow to change what is displayed and printed during the minimizations. The `loops` ruler determines the number of iterations that should be

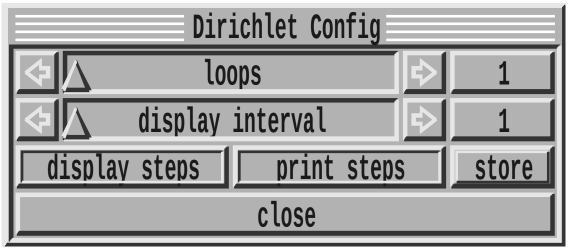


Figure 8.21: Config-Layer for the Dirichlet-Minimizers

applied, if its value is n and a minimization method is applied to M_i the result is M_{i+n}. If the `print steps` checkbox is selected then for each step of the CG method the step number and the residuum are printed to the shell **GRAPE** was started in. The `display steps` checkbox and the `display interval` ruler determine which steps of the CG method should be displayed — every `display interval`th step if the checkbox is on.

The configuration layer can of course be closed with the `close` button, but it is a good idea to keep it open during the minimization since most of the options can still be changed while the minimization is running. If for example displaying each step slows down the minimization too much it can be disabled completely by switching off the `display steps` checkbox or the number of steps displayed can be reduced with the `display interval` ruler.

The `store` checkbox allows to enable a special feature: if it is on the surfaces M_i of the minimization are stored as frames of a `Surface` instance, the minimization process then can be played back as a video (see section 4.3.5.2). This only works for the `-step` methods on a `Surface` instance, the minimization is always applied to the last frame and the result is added as a new frame at the end of the sequence.

When the minimization is started all menus and layers but the configuration layer are disabled and a layer containing information about the minimization process (figure 8.22) is opened. It

Figure 8.22: Info Layer for the Dirichlet-Minimizers

shows the frame and loop, the object (there can be two objects at each frame, i.e. `TimeStep`, the `pre_object` and the `post_object`) that is currently minimized and the step and residuum of the CG method. The minimization process can be suspended temporarily with the `Stop/Cont` checkbox or aborted completely with the `Abort` button.

8.2.7.3 Creating Initial Triangulations

The *Dual* project defines methods for class `Geom2d` (for single geometries) and `Surface` (for parameter-dependent sequences of geometries) therefore it adds the *Surface* project. Initial surfaces for the *Dual* minimization methods can for example be created with project *Explicit* (see 8.2.5) or *Build* (see 8.2.4 and the example in section 8.2.9).

The *Build* project allows to create topological surfaces of arbitrary type and genus, it especially allows to control the boundary restrictions. To simplify the creation of surfaces with boundary restrictions the minimization algorithms try to detect the boundary types automatically: if a boundary line is straight or planar it is assumed to be restricted to the straight line or plane, if some other restriction is needed it has to be specified explicitly.

The *Dual* project provides an interface (see figure 8.20) to some methods for modifying triangulations, together with the methods from the *Refine* project described in the next section they allow to create very good initial triangulations for the minimization methods (please read section 8.2.9 before you use these methods):

`equiang` This method implements a Delaunay swap algorithm for triangulations in three-space (compare 6.3.4.5), together with

`mollify` which smoothes the triangulation by averaging the points it allows to make triangulations more uniform and regular. By using `mollify` as a display method (i.e. press `disp` `mollify`, to switch back press `disp` twice) you can configure how often `"mollify-send"` should be applied and whether boundary points should be mollified, too.

There is also a `regenerate` method which removes degenerated triangles, this might be necessary if for example an edge of a surface patch created with the surface builder is degenerated or if triangles degenerate during the minimization process. But be careful, the resulting triangulation may be singular and you might not be able to continue working with it, saving the `Surface` with `write *.gr` before applying `regenerate` might be a good idea.

8.2.8 Project *Refine*

The *Refine* project provides an interface to some interactive refine methods, the most important one is the curvature dependent refinement. The methods for selecting edges for refinement are defined on `Adapt2d`, the refine methods themselves are based on the methods described in section 6.3.3.4 and provide special support for `Surface`s.

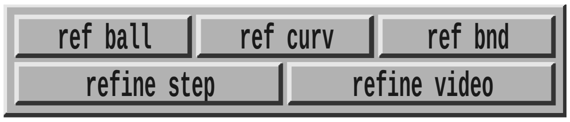


Figure 8.23: User-Interface for the *Refine* Project

Refining a step of a `Surface` changes the discretization of the step itself and possibly some neighbour steps therefore the refine methods have to be applied carefully, please read the next section before you use them.

There are three different display methods (you have to press `disp` first, then the `ref ...` button) for selecting which edges should be refined. The display is switched to grid mode and the geometry is rendered in green, the edges that will be refined are drawn in red and the edges that will be inserted are drawn in yellow (the result can differ slightly from what you see because the refine algorithm may add some correction steps).

`ref curv` — With this method triangles in regions of high Gaussian curvature can be refined (see the next section for an example), all edges with a curvature higher than the value of the ruler added to the option menu are selected for refinement. The curvature of an edge is computed from the angle(s) between the surface normals at opposite vertices of the triangle(s) the edge belongs to (the normal rotation along and if possible across the edge).

`ref bnd` — Refine the triangles with a boundary edge with a length greater than the value of the ruler added to the option menu. As for the other methods the longest edge of the triangle is selected for refinement, therefore it may be necessary to refine triangles twice to really refine the boundary edges.

`ref ball` — Refine all triangles which are inside a ball and have an edge longer than the selected sidelength value. The center of the ball can be changed by pressing the `ref ball midpoint` button and picking a point in the graphics window.

Be careful: These methods replace the `"set-rfc-field-send"` method with their own versions, if you want to use the original method after using one of these methods you must restore `adapt2d_set_rfc_field_send` as `"set-rfc-field-send"` method.

Refining all steps of a `Surface` instance with `refine-video` is only possible if they all have the same discretization, therefore you can only use this method before methods like `refine-step` and `equiangulate` which change the discretization locally. `refine-step` only refines the current step of a `Surface` instance (and its neighbour steps), the objects of the steps are split if necessary (see the next section for further information).

In some cases refining several triangulations with the same discretization using the same refine field results in triangulations with different discretizations. This can only happen if the refine algorithm has to add correction steps for refining the marked edges and if corresponding triangles have different longest edges, therefore this problem is more likely to occur when a whole sequence of steps is refined with `refine-video`. You can still continue working with these triangulations, but you won't be able to interpolate them or to refine them again.

8.2.9 Project *Diri*

Project *Diri* combines the projects *Build* (8.2.4), *Dual* (8.2.7) and *Refine* (8.2.8), it allows to create in an easy way initial surfaces, to minimize them and to apply conjugation.

We use the CLP example from section 8.2.4.2, adding the project *Diri* with `add project`, switching to the `"build"` object and reading in the `CLP.bd` input file with `read *.in` from the option menu creates a sequence with five frames, three of them are shown in figure 8.18. Applying `merge` and `close` removes the boundary edge between the two quadrilaterals and creates a single geometry with six straight boundary lines.

First we explain how to modify a single geometry to get a good initial surface for the minimization, handling sequences is discussed later. Select frame 3 (the one with the highest symmetry) either with the `Frame` input field or the arrows, you should see top left surface in figure 8.24.

The first method we use is `"mollify-send"` which smoothes the surface (in fact all frames of a `Surface` instance) by replacing each interior point by the average of its neighbours. Pressing `mollify` applies this method not only once — this would only affect the points along the common edge of the two quadrilaterals — but several (by default 10) times.

The second useful method is `"equiangulate-send"` which implements a Delaunay swap algorithm for surfaces in three-space (see 6.3.4.5 for further information). Be careful, the `"swap-edges-send"` method from the *tr2d-edit* project also implements a Delaunay swap algorithm but it can only handle flat triangulations, and like most methods from the *tr2d-edit* and *grid-edit* projects it doesn't support `Surface` instances. Now press `equiang` and again `mollify` to smooth the result from the `"equiangulate-send"` method, the resulting surface is shown in the middle of the top row in figure 8.24.

By now the triangulation is already better than it would get by just using the minimizing methods but obviously we need more triangles in the middle of the surface and especially at the two vertices where the quadrilaterals met. These are the regions with the highest Gaussian

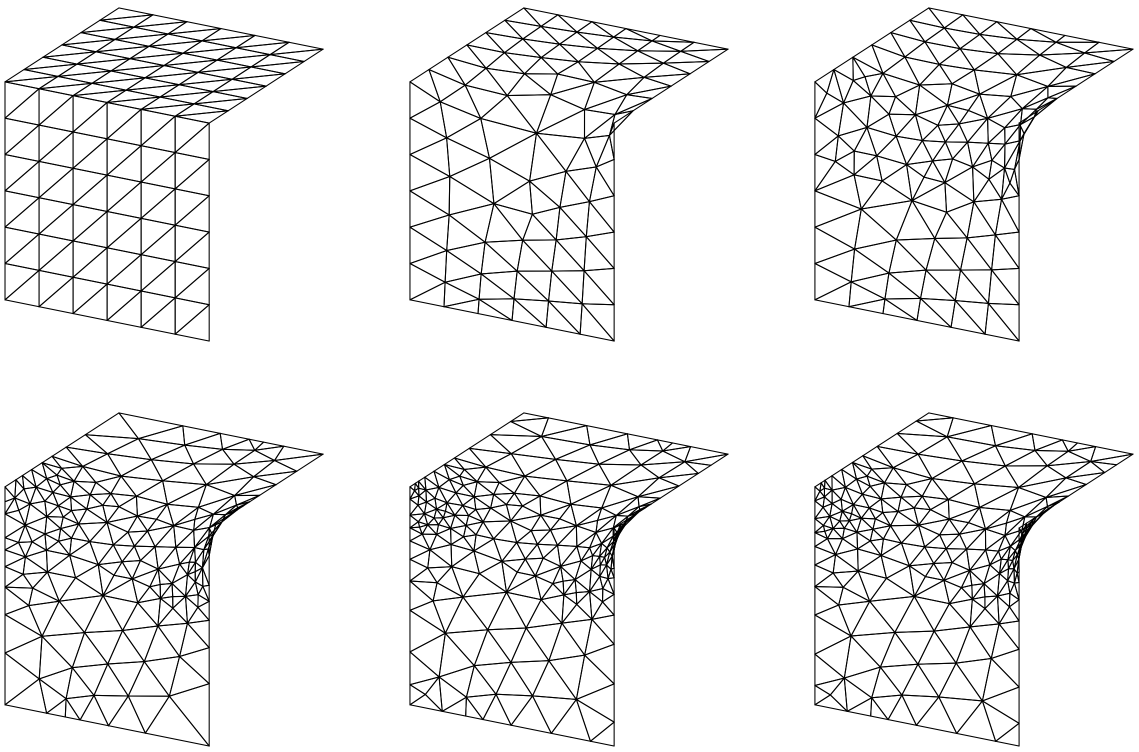


Figure 8.24: Creating a triangulation for the CLP example. Starting with the two quadrilaterals (top left) and applying mollify, equiangulate and refine methods we get the initial surface for the minimization (bottom middle) which is almost identical to the minimized surface (bottom right).

curvature therefore we can use the curvature dependent refinement `disp` `ref curv` to insert triangles exactly were they are needed. Change the curvature ruler value to `0.25` to select some more edges and refine them with `refine step`. Applying `mollify` `equiang` twice and one more `mollify` to smooth the result of the refinement gives the top right surface of figure 8.24.

Refining the surface again with `refine step` inserts more triangles at the two vertices, now the triangulation is fine enough to allow `"mollify-send"` to also move the boundary points. Press `disp` `mollify` and switch on the `mollify boundary` checkbox (by changing the mollifies ruler value you could also change the number of times `"mollify-send"` is applied) and reset the display with `disp` `disp`. Now you get the bottom right surface in figure 8.24 by using `mollify` `equiang` twice and `mollify` again.

To remove the larger triangles at the boundary we use `disp` `ref bnd` with a value of `0.2` and `refine step` `mollify` `equiang` `mollify`. One more refine step `disp` `ref curv` `refine step` and some final smoothing with `mollify` `equiang` twice and `mollify` gives the middle surface in the bottom row of figure 8.24, the initial surface for the minimization.

Now apply the minimization algorithm by pressing `CG-step` several times (applying it five times yields the bottom right surface of figure 8.24). Even the first time the points only move very little since the initial surface already was close to the minimum! This is the case because `"mollify-send"` and `"equiangulate-send"` make the triangulation more regular and uniform, therefore they have a minimizing effect (see the minimality condition in section 8.2.7.1).

When you look at the frame sequence of the CLP `Surface` instance using the scene time ruler named `Surface_Build "CLP"` in the manag menu you will notice that the first and the last frame (times 0 and 1) still have the starting discretization, only `"mollify-send"` has been applied to them. At the second and fourth frame (times 0.25 and 0.75) the discretization changes to that of the third step (time 0.5) we have been working on.

Methods like `"refine-send"` and `"equiangulate-send"` which change the discretization of the triangulation they are applied to must also exist on class `Surface` to handle the discretization changes, if they were just applied to all triangulations of the sequence the interpolation wouldn't work afterwards. Of course all triangulations could be changed in the same way but this doesn't make much sense for example for the refine methods, see figure 8.26.

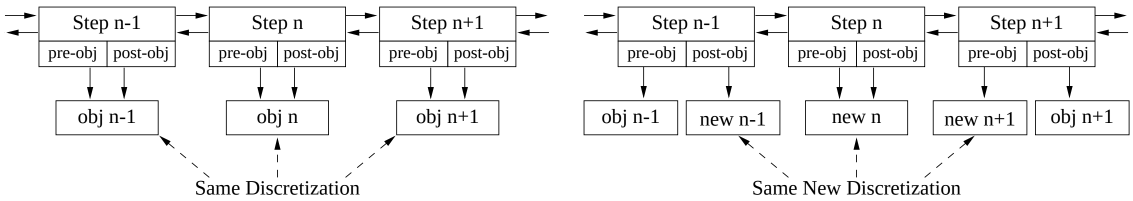


Figure 8.25: Local discretization change. Starting with a sequence of steps with identical discretizations (on the left) and applying a discretization changing method to step n results in a sequence with split objects at steps $n-1$ and $n+1$ (on the right).

Let's assume the steps $n-1$, n, and $n+1$ of a `TimeStep` sequence (which is used by `Surface` to store the triangulations) have the same discretization. If the discretization of step n is changed that one of the neighbour steps $n-1$ and $n+1$ has to be changed in the same way, otherwise these steps couldn't be interpolated with step n any longer. To prevent the spreading of the discretization change to the whole sequence the `pre_object` and `post_object` of the steps $n-1$ and $n+1$ are split and only the discretization of the `post_object` of step $n-1$ and the `pre_object` of step $n+1$ which are used for the interpolation with step n are changed, see figure 8.25. The old objects are still available as `pre_object` of step $n-1$ and as `post_object` of step $n+1$, they are used for the interpolation with steps $n-2$ and $n+2$. It doesn't matter if one of the steps is already split, for step $n-1$ or $n+1$ the splitting then is just skipped, if step n is split the two objects are handled in the same way but separately.

Each step that is split needs additional memory and makes the sequence more complex. Therefore the best way to handle sequences is to apply the mollify, equiangulate and refine methods only to every second frame, using an odd number of frames is also a good idea, then the methods can be applied to the first and the last frame. In the CLP example with five frames we can apply the methods to frames 1, 3 and 5 to get the surfaces shown in figure 8.26, the discretization then only changes at the frames 2 and 4.

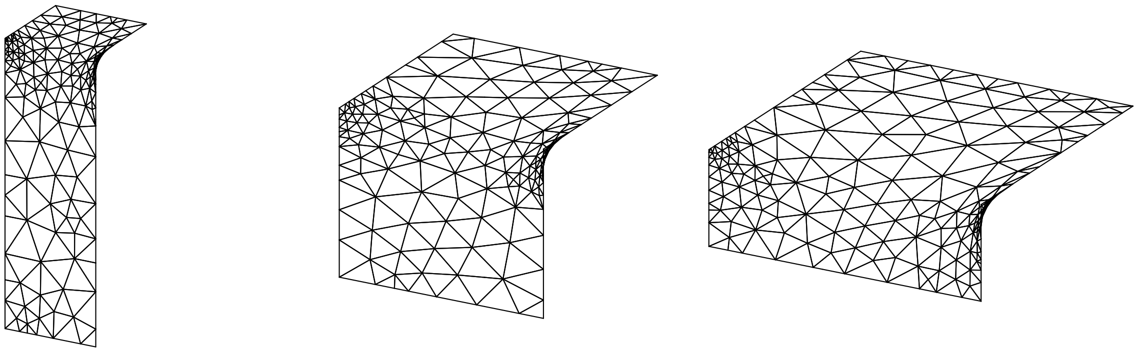


Figure 8.26: Three minimized frames from the CLP sequence. Each frame has a different discretization that was created using the curvature dependent refinement method which refines triangles in regions of high Gaussian curvature.

Of course a single triangulation can be conjugated with the `dual step` button and viewed by changing the display method to `"conjugate-disp"`, but to be able to reflect the conjugated surface all frames have to be computed and the current geometry of the `Surface` instance has to be changed with the task button to `TASK: CONJUGATE`. Pressing the `reflect` button allows to select boundaries for the reflection (see section 8.2.3.2), reflecting the minimized example surface twice gives the sequence in figure 8.27.

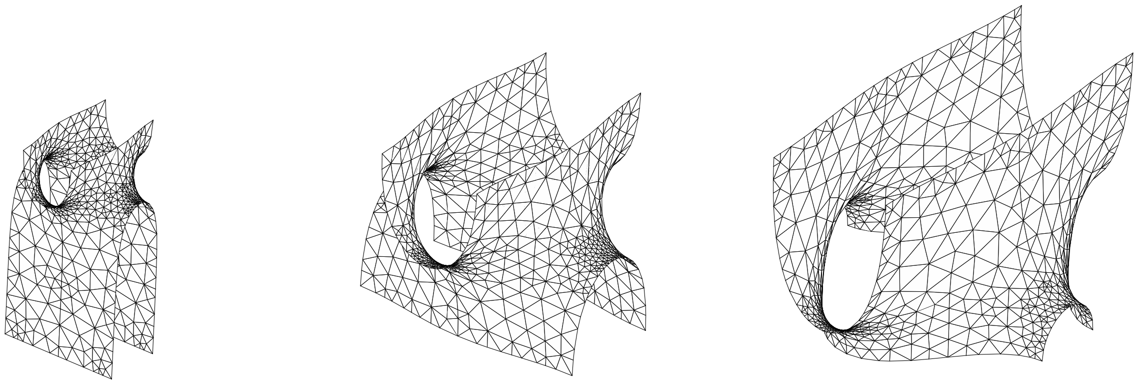


Figure 8.27: Conjugated and reflected surfaces from the CLP sequence.

8.3 Misc Projects

8.3.1 Project *Model*

The project *Model* contain some tools to modify directly the geometry such as scaling, translating, and rotating. This really affects the geometry and not only the transformation matrices.

When you create your own geometry classes you have to provide a `"model"` method to be able to work with this project on your geometries. The method gets the transformation matrix as parameter and has to performs the transformation.

8.3.2 *RGBVideo*

This project allows to create arbitrary size bitmap files or video sequences. The file handling is done by the hidden project *Images*.

Currently this project only supports `GL` graphics.

8.3.2.1 *Images*

All created images will have the same name, the extension will be correctly set; when more than one image is written the file names contain automatically incremented numbers. The file name can be changed by setting the environment variable `GRAPE_IMAGE_NAME`, if not set a warning is displayed. The default name is `grape`.

Internally this projects implements an abstract pixmap class with a subclass for every supported file format. So new file formats can be added easily; unfortunately the programming interface is still undocumented (but will be documented sometimes).

The only button is

`SGI-Image`/`Targa-File` Choose the file type to use.

- **SGI-Image:** SGI RGB files. This format requires the SGI image library. It is not supported by non SGI machines.
- **Targa-File:** Uncompressed targa files.
- **ppm:** Standard portable pixmaps. Decide by a `Checkbox` if you want the raw (smaller and faster) version of this format or the ASCII version (more portable).

8.3.2.2 *RGBVideo*

The *RGBVideo* user interface has its standard place in menu `opt 4`. The upper half of the `Group` deals with video related options and the activating buttons. The lower part controls image sizing and control rulers.

`main time` Main time indicator, which is linked with the manag menu maintime. Choose start times for previewing here.

`start time` Start time of video-scene.

`end time` End time of video-scene.

`# frames` Total number of frames to be written.

`% preview` Only show `value` frames in preview mode. `1` means: all frames are shown.

`<` Backward preview.

`>` Forward preview.

`REC` Start video recording from start time to end time. A wait layer will pop up which indicates the progress in a % bar.

Caution: Moving the mouse inside the graphics window or overlapping windows destroy the images.

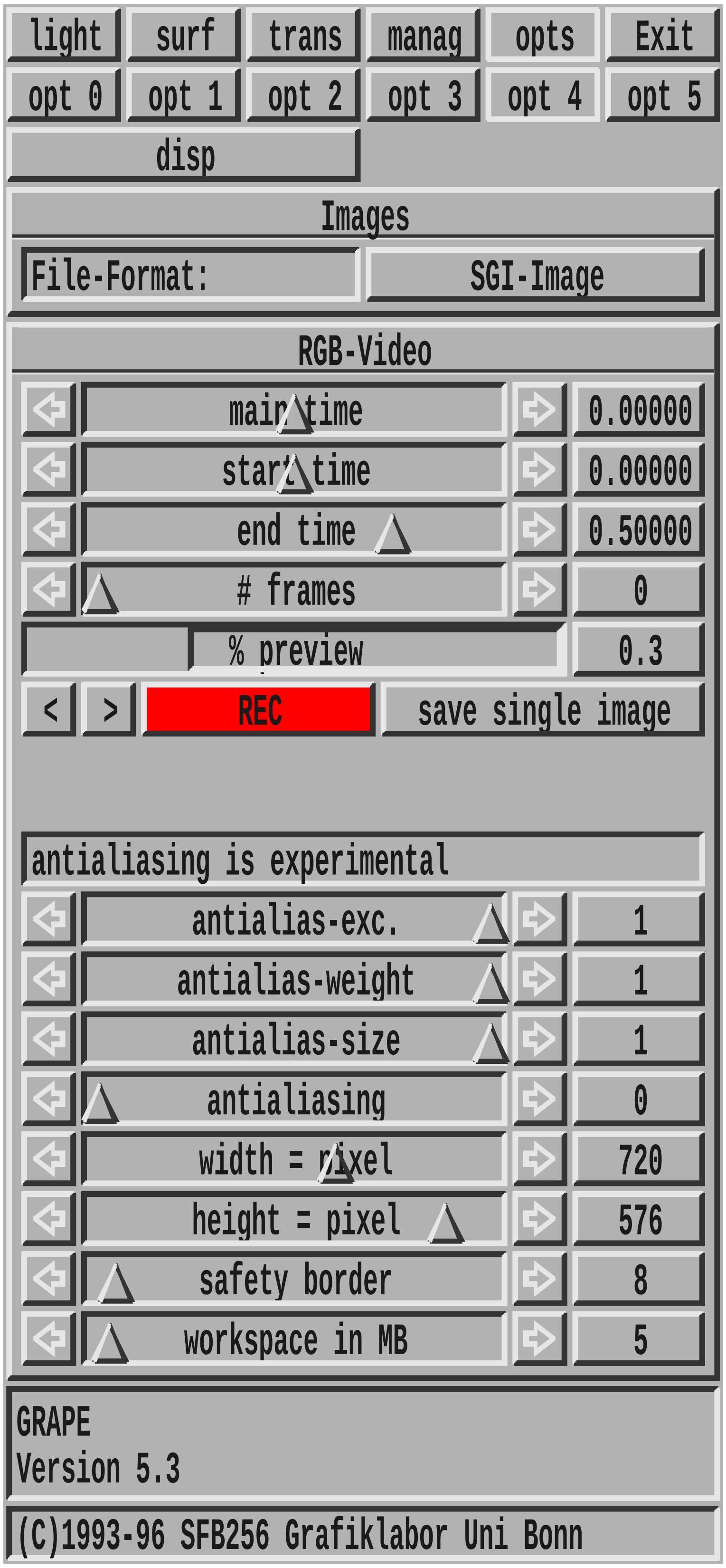


Figure 8.28: User Interface of Project *RGBVideo*, Containing the *Images* Part

`save single image`

Save one single arbitrary sized image. The image may be displayed in segments if it does not fit to screen. Check the output files: sometimes there are errors like gray lines. Before drawing the image you have to select a region. A rectangle appears in the graphics window, pull the nodes to resize it.

The mouse buttons are described in the main message area.

left: Drag the corner nodes and the edge middle nodes to move them. Click into the center to adjust the size with mouse movements. Click and drag somewhere else to move the frame.

The aspect ratio is constant. It is calculated from the image dimensions (the image pels are squares).

middle: Cancel. The region values are reset to the previous state.

right: Accept the region and create the file. The drawing process cannot be stopped by the user.

ESC: Exit the region editor. The values are preserved.

SPACE: Reset the region values to the previous state.

C-SPACE: Reset the region to the default values.

The mouse has to stay inside the graphics window to accept clicks.

There is no antialising for single images because there is no size limit. You can write a single image with `n`∗size and apply a scaling filter afterwards.

Antialiasing for video sequences. This option is a bit experimental and does not work properly on all machines. It is optimized to reduce flickering. `n` points on an ellipse around the real value are sampled.

`antialias-exc.`

Aspect ratio of the ellipse.

`antialias-weight`

Weight of the extra pixel values in respect to the center value.

`antialias-size`

Distance between center and extra pixels. The unit is screen pixel distance.

`antialiasing`

Number of extra pixels. `0` means antialiasing is off. Good values are > 4.

`width = pixel`

`height = pixel`

Enter the output file size in pixels. The default values are PAL resolution.

`safety border`

GL optimizes drawing patches by omitting partly seen ones at the border. Leave a safety border to reduce this effect if image segments are merged.

`workspace in MB`

Writing files needs a lot of memory. The more space available the bigger the chunks the faster the program — if it fits into real RAM and no swapping occurs. There is no C-function to get a proper value, enter the size of free memory here.

The default value is 5MB which should be available on most machines.

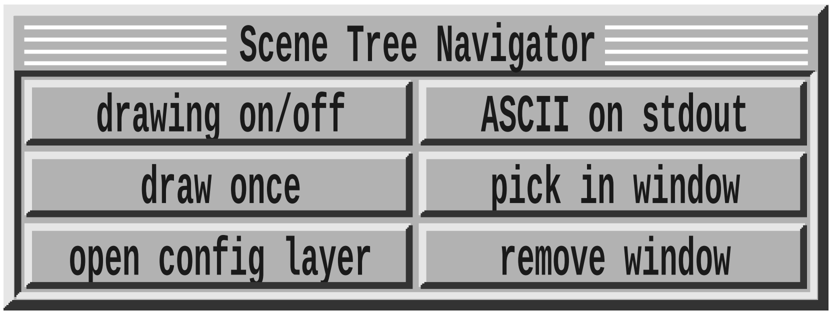


Figure 8.29: Main User Interface of Project *Scene-Tree*

8.3.3 Video Recorders and GRAPE

The design goal for making movies with **GRAPE** was to provide a simple interface to be able to control all of the recorder's functions. In addition, the realization of movies should be as easy as possible. Currently there are four additional mouseclicks necessary to record a given **GRAPE** timescene onto the video disc recorder.

Since special hardware is necessary this feature is not included in standard libraries. For further information contact your distributor.

8.3.4 *Scene-Tree*

This project tries to display a global view of the data hierachy tree. This is useful for **GRAPE** novices which are not sure about the object buttons in the manag menu, and it allows to quick navigation in a complicated data hierarchy.

`drawing on/off` Toggle updating the data hierarchy tree window every time the data hierarchy tree changes. This switches on the window as well.

`draw once` Draw the data hierarchy tree once. If the data hierarchy tree window hasn't existed before, it is created. The window will not be updated later on.

`open config layer` Create the configuration layer for the data hierarchy tree window. You should close the layer before removing the project.

- `show nextlinks` Toggle drawing of pointers like Scene->next_scene.
- `show downlinks` Toggle drawing of pointers like Scene->object.
- `show backlinks` Toggle drawing of connection to left pushed objects.
- `show equals` Toggle drawing of lines between equal objects at distinct positions.
- `mark scenes` Toggle drawing Scenes in white color.
- `draw name` Toggle drawing name at object position.
- `start from root` **On:** tree is scanned from the root node

  Off: tree is scanned from current object.

`depth:` Max number of followed downlinks.

`back:` Go back **n** steps from current object before scanning tree.

`total depth:` Max number of followed downlinks+nextlinks.

`dismiss` Close this layer.

`ASCII on stdout` Scan the data hierarchy tree from its root. Print the result on stdout.

`pick in window` Pick in the tree window.

left mouse: goto object, as if walked in the local view in the manag menu

middle mouse: quit picking

right mouse: open an information box about the picked object and temporarily show its name in the tree window

C-l: redraw data hierarchy tree window

r: goto root and redraw window

`remove window` Remove the data hierarchy tree graphics window.

The project adds the display method `"draw-scene-tree"` as well, which allows to zoom and rotate or to print the tree; the user interface operates with send–methods found at class `Root`.

8.3.5 CoordSys

The Project CoordSys is used to enable multiple coordinate systems in the Scene Tree. These coordinate systems are independent to each other and independent to the global coordinate system, configured in the trans menu. The coordinate systems in the scene tree can be saved and loaded using the `write gr/read gr` in the manag menu. The project will add a coordsys at the current scene tree position and add one button to the first option menu.

`Edit 'coordsys' (Coordsys)` This button is used to open the configuration layer for the specific coordinate system. The first coordinate system found in the scene tree is used.

The Configuration layer of the coordinate system allows various options to configure the coordinate system for your application. Some interactives are the same for all three directions and are describted only one time.

`set-3-sys` Configure the coordinate system as a default three axis coordinate sytem.

`set-boundary-system` Configure the coordinate system as a default 12 axis coordinate system.

`Min:`, `Max:` Set the minimum and maximum values in the specific direction. The coordinate systems axis will last from the minimum to the maximum value in that direction.

`Marks` How many mark should be set at the axis in the desired direction. If a mark is near a crossing of two axises the mark will be suppressed, but if the value of this mark was visible it will be replaced by the crossing value.

Label The name of the axis, which is printed on top of the axis. The lable will only be show together with the arrow at the end of the axis.

Expert Open the special configuration layer for the desired direction. This layer includes the following configuration options:

- **Enable** Enable or disable the different coordinate axis. For each direction five axis are possible. Four at the position of the minimum and maximum values in the orthogonal directions. One variable axis anywhere betweeen these values or even outside.
- **Marks** Enable or disable the marks at the five axis. The axis obviously must be enabled.
- **Labels** Enable or disable the value printing at some marks and label output at the top of the axis. The values are only printe if the marks are enable at the specific axis. The same with the label itself and the arrow.
- **Arrows** Enable or Disable the arrow at the top of the axis. If the maximum value is lower than the minimum value the arrow will be printed at the bottom, but will always direct in the maximum direction.
- **x-Trans**, **y-Trans**, **z-Trans** Translate the value text orthogonal to the axis.
- **First Mark** Which of the marks counted from minimum to maximum value should be the first one with a value given.
- **Skip Mark** How many marks should be skipped while printing values.
- **Format:** Print format for the value output. Do not include a or a similar format in this field, because this migh crash your program.
- **dismiss** Close the expert config layer.

Radius Secify the radius relativ to the axis length of the longest axis.

Mark-Factor Specify the radius of the marks relativ to the radius of the axis.

Arrow-Size The radius of the Arrows relativ to the radius of the axis.

Arrow-Distance How much space should be left between the end of the axis and the beginning of the arrow.

x-Direction, **x-Direction**, **x-Direction** The relativ or absolute position of the variable coordinate axis in the specific direction. If any of these values should be an absolute value the button **Abs-Pos** in that direction should be enabled.

`Discret`	Number of patches for the axis in rotation direction. Increase this value if the axis is not smooth enough.
`End Rounded`	Using this option the end of the axis is rounded instead of cutted of. Especially a boundary coordinate system will look much better if you enable this option.
`dismiss`	Close the specific configuration layer.

Chapter 9

Interactive Classes and Devices

9.1 Interactive Classes

The handle consists of an own complex class structure with some abstract classes, but is easy to use if once understood. To make things even more easy there is a special interface (see 5.5.3) for some classes; if you want to change an existing `Item` you have to use the methods.

9.1.1 General Structure

9.1.1.1 Menu concepts

The manager has got a list of layers (see 9.1.3.16), where each layer is a independent part of the manager. Each layer, which is a subclass of group (see 9.1.3.12), can contain several objects or is a total menu itself. A layer is added to and removed from the manager with the methods `"add-layer"` 9.1.3.17 and `"remove-layer"` 9.1.3.17. The layers of the handle are comparable with the windows of graphical user interfaces like X11 and the layers have a similar using mechanism: Each one can be moved, sized and iconized. The menus of **GRAPE** are implemented as groups (9.1.3.12), which must be inserted into a layer of the manager. A group contains several items, but it can be handled like one single item from the superior groups and layers. If you start up a **GRAPE** program the standard manager consists of single layer which contains only a selector and a textmessage. The selector will then add the group, which contains all control-elements of the manag-menu (see 4.3.4), to the layer. If a different standard-menu is selected, for example the trans-menu (see 4.3.8), the selector will remove the manag-group from the layer and will add the trans-group. Pay attention that no different layers are used up to now. A layer is part of the menu, which will overlay the normal menu, for example the layer, which contains the project-add list or the file lists. If a layer depends upon another item of the normal menu structure it will be removed as well if the item is removed from the menu structure and will be added again automatically.

9.1.1.2 Adding and Removing of Items

Each item must be added to a group or layer with the method `"add-inter"` (see 9.1.3.12) of the group or layer. The item can also be a whole group as it was described in the last sub-subsection. The group will try to add the item with the size, which can be set using the method `"set-pref-size"` (see 9.1.3.15), and the position, which can be set using the method `"set-pref-pos"` (see 9.1.3.15). (see also next subsection.) The position arguments must be given relatively to the group, because you don't know which position the group will have. If an item doesn't fit into the group at the preferred position, the group will try to add it at another position. If this fails either, the group will try to find a place for the item in one of the group's replacement groups, which can be defined using the methods `"add-replacement"` (see 9.1.3.12) and `"remove-replacement"` (see 9.1.3.12). You must remove the item always from the origin group, also if it was replaced, using the method `"remove-inter"` (see 9.1.3.12). The replacement groups are totally transparent from the sight of the programmer.

Each item, which is added to a group, which is added to a group, ..., which is added to a layer, which is in the list of the manager, is called visible. An item is called visible even if it is totally obscured by a layer. The visibility state of an item can be determined using the method `"is-visible"`. Each item has got two methods, which are called if the item is added to the manager or if it is removed from the manager. These methods can be changed using `"set-add-action"`. These methods are also called if the group or a superior group of the item is removed from/added to the manager.

9.1.1.3 Size and Position of Items

Each item has got a pref-pos and a pref-size. These values, which are set using `"set-pref-pos"` (see 9.1.3.15) and `"set-pref-size"` (see 9.1.3.15), are only the position and size the item will prefer. The item can also be at different position, if the pref-pos is already occupied by another item. It is also possible to have the item positioned and sized automatically. (see description of the methods `"set-pref-pos"`, `"set-pref-size"`, `"set-fill-mode"` (see 9.1.3.15)). It is possible to use these automatic function independently for the two directions, for example x-size, x-pos and y-size fixed, y-pos automatically or y-values fixed, x-values automatically. If the method `"calc-inter"` (see 9.1.3.12), is sent to a group, the group will resize and reposition all its elements. In this mechanism the group will first place all items which have both positions fixed. After that all items, which have at least one automatic position are placed. In the `"calc-inter"` method, the group will also try to place the item, which have been replaced to another group. After all possible items have been placed, the group will try to place all items, which couldn't be placed up to now, in one of the replacement-groups. (about replacement refer to the last subsection.) Pay attention, that this mechanism is different to the one used if you add item per item.

9.1.1.4 Event handling

The manager gets event per event from the control-device. These events can be of different types, for example graphic- or control redraw, user-actions like mouse and keyboard and so on. The manager will start the necessary action. If the event was a user-event like mouse or keyboard, the manager will search the concerned item and call its handle-event method. These different methods of the handle classes will handle the event and they must not wait for the next event, because then the will lock the event queue and redraw events can't be handled correctly.

9.1.1.5 Redraw Mechanism

It is strictly forbidden for the items to draw themselves, as they have done in old versions of **GRAPE**. Drawing must be done by the manager calling the `"check-refresh"` (ref. 9.1.3.17) method on its layers and so on their group(s). If an element needs to be drawn, for example a button which was pressed down, it must use the method `"set-refresh"` (see 9.1.3.15), to mark itself. The manager will search for all items, which must be redrawn, and redraw them, before trying to get the next event from the control-device. Using this mechanisms also partial hidden object, who lay behind another layer, will be redrawn correctly. If the used control-device supports clipping, like the X11-device, these ability will be used. If a whole area of the manager must be redrawn, for example if you move or close an upper layer, the method `"mark-refresh-region"` is used to search and mark all items automatically. Groups and layers have got different refresh-flags for their items and for themselves (the border, title and the place which isn't occupied by any element).

9.1.1.6 Actions

Interactives will send the action method to themselves, if you use them, for example by pressing a button, finishing typing in an input, moving a ruler and so on. You can set this method using the method `"set-action"` (see 9.1.3.14). Normally this method is the `"default-action"`. If you change this action method, you can change the behavior of the elements.

9.1.2 Class Hierarchy

`Manager:` connection between devices and GRAPE, receives events and initiates the associated actions, contains layer management and display method ruler management. (9.1.3.17 `Manager`)

`Item:` abstract superclass for all visible objects in the handle which implements size, redraw and position management and general functionalities, allows to be grouped. (9.1.3.15 `Item`)

- `Rectangle:` colored rectangle. (9.1.3.21 `Rectangle`)
- `Textmessage, StaticText:` static text and messages. (9.1.3.29 `Textmessage`, 9.1.3.27 `Statictext`)
- `Interactive:` abstract superclass for all interactive objects in the handle, e. g. clickable objects. Implements event handling. (9.1.3.14 `Interactive`)
  - `Input:` input field for numbers or single lines of text. (9.1.3.13 `Input`)
  - `Button:` starts action if clicked. (9.1.3.2 `Button`)
    - `XList_Button:` opens a layer containing an `XList`, `FileList` or `Methlist`. (9.1.3.31 `XList_Button`)
    - `Checkbox:` toggles a flag. (9.1.3.4 `Checkbox`)
    - `Cycle_Button:` button with switching label. (9.1.3.7 `Cycle_Button`)
      - `ComboButton:` button with selectable label. (9.1.3.6 `ComboButton`)
    - `Step_Button:` autorepeating button. (9.1.3.28 `Step_Button`)
  - `Group:` rectangular container for `Items`, which forwards some methods to them. Since `Group` is a subclass of `Item`, `Groups` can be nested. (9.1.3.12 `Group`)
    - `Layer:` a movable overlay `Group`. (9.1.3.16 `Layer`)
      - `MethLayer:` special `Layer` for choosing display methods. (9.1.3.19 `MethLayer`)
    - `Bar1d:` abstract superclass for `Ruler`, `Slider`, `Scrollbar`. Edit scalar values with the mouse or by typed input. (9.1.3.1 Bar1d)
      - `Ruler, Slider:` ruler and slider. (9.1.3.22 `Ruler`, 9.1.3.24 `Slider`)
        - `Function_Bar1d:` abstract superclass for a `Bar1d` controlled by a spline. (9.1.3.9 `Function_Bar1d`)
        - `Function_Ruler, Function_Slider:` different looking versions of `Function_Bar1d`. (9.1.3.10 `Function_Ruler`, 9.1.3.11 `Function_Slider`)
      - `Scrollbar:` a scrollbar with bar size management. (9.1.3.23 `Scrollbar`)
    - `Spline_Editor:` interactive editor for splines. (9.1.3.26 `Spline_Editor`)
    - `XList:` table of choosable strings and an input for arbitrary strings. (9.1.3.30 `XList`)
      - `FileList:` file selector. (9.1.3.8 `FileList`)
      - `MethList:` method selector for the class tree. (9.1.3.20 `MethList`)
    - `CheckField:` easy to use grouping class for `Checkbox`es, you can define complex connection rules. (9.1.3.3 `CheckField`)
    - `Color_Sel:` color selector for RGB and HSV color model. (9.1.3.5 `ColorSel`)
    - `Matrix_Edit:` interactive editor for some parts of transformation matrices. Used in trans-menu, light-menu, ... (9.1.3.18 `Matrix_Edit`)

`Spline1d:` spline. (9.1.3.25 `Spline1d`)

9.1.3 Classes and Methods

This is an alphabetic list of the user relevant classes and methods. The class structures are not shown because all accesses have to be done by methods.

9.1.3.1 `Bar1d`

(subclass of 9.1.3.12 `Group`)

`Bar1d` is an abstract superclass which implements common functionality of `Ruler`, `Slider`, `Scrollbar`, `Function_Bar1d`, `Function_Ruler` and `Function_Slider`. You cannot create a single `Bar1d` instance (since the class is abstract), but only instances of the listed subclasses. Methods for things like bounding the edited value, scaling the ruler, slider coloring and fixing of the `0` position are documented here.

The `Bar1d` validates the min-max-values for the variable that is seen in the input field.

```
typedef enum declaration_format{
  dfInt,
  dfFloat,
  dfDouble,
  dfString,
  dfRegexp
} DECLARATION_FORMAT;

typedef enum orientations {
  orHorizontal,
  orVertical
} ORIENTATIONS;

/* Internal use only. */
typedef enum bartypes {
  btRuler,
  btSlider,
  btBoundedRuler,
  btFunctionRuler,
  btScrollBar
} BARTYPES;
```

GRAPE(bar1d, "fix-min-max")(fix_min, fix_max)

> **BAR1D *bar1d**
> **int fix_min, fix_max**

If one of `fix_min` and `fix_max` is `ON`, the corresponding value bound of the Bar1d cannot be edited by the user in the configuration layer. If only one of them is `ON` and the other is `OFF`, it remains possible to set `min < max` to remove all bounds.

This option is turned off by default to allow later changing of wrong values. It should only be used in very special cases.

GRAPE(bar1d, "set-bar-color")(rgb)

> **BAR1D *bar1d**
> **VEC3 rgb**

Set the color of the moving bar to the RGB color `rgb`.

GRAPE(bar1d, "set-bar-color-index")(color_index)

> **BAR1D *bar1d**
> **int color_index**

Set the color of the moving bar to the indexed color `color_index`.

GRAPE(bar1d, "set-eval-fn")(eval, inv_eval)

> **BAR1D *bar1d**
> **double (*eval) double**
> **double (*inv_eval) double**

Set the evaluation functions. Every time the variable is read or written the according function is applied, `eval` before writing and `inv_eval` after reading.

As you see the functions are `double (double)`, the type conversion (`int`, `double` ...) is done automatically.

GRAPE(bar1d, "set-min-max")(min, max)

> **BAR1D *bar1d**
> **double min, max**

Set bounds for the edited value, if the value is out of bounds it will be corrected to the nearest one. `min > max` means no bounds.

GRAPE(bar1d, "set-offset")(offset)

> **BAR1D *bar1d**
> **double offset**

Set the value represented by the left or top position of the ruler, for example: for a ruler, which has min = 3.0, max = 4.0 and a scale of 0.5, you should set an offset of 3.0 or 3.5 to use the ruler optimally. If you set the offset to a value, which will cause the position of the ruler to be out of the possible range, the offset-value will be corrected automatically.

GRAPE(bar1d, "set-print-format")(format)

> **BAR1D *bar1d**
> **char *format**

Set the input field's print-format (see 9.1.3.13 Input).

GRAPE(bar1d, "set-scale")(scale)

> **BAR1D *bar1d**
> **double scale**

Set the ruler scaling. `scale` means the difference between the value represented by the handle being at the most left position and being on the most right position.

GRAPE(bar1d, "set-step-size")(stepsize)

> **BAR1D *bar1d**
> **double stepsize**

You can step the Bar1d by clicking on one of the arrow Buttons. Set the size of this steps here.

GRAPE(bar1d, "set-type")(type)

> **BAR1D *bar1d**
> **DECLARATION_FORMAT type**

Set a new type for the variable. You can do this together with setting the new variable (with the new type) by `"set-var-and-type"`.

GRAPE(bar1d, "set-value-changed")(value_changed)

> **BAR1D *bar1d**
> **char *value_changed**

Set a method the Bar1d sends to itself every time its variable's value has changed.

GRAPE(bar1d, "set-var-and-type")(var, type)

> **BAR1D *bar1d**
> **void *var**
> **DECLARATION_FORMAT type**

Set a new variable to manipulate interactively with the Bar1d together with its type. Setting both at the same time avoids several problems.

GRAPE(bar1d, "set-variable")(var)

> **BAR1D *bar1d**
> **void *var**

Set a new variable to be interactively manipulated by the Ruler/Slider/Scrollbar.

GRAPE(bar1d, "update")()

> **BAR1D *bar1d**

Overloading inherited method of class `9.1.3.14 Interactive`. This method will will fetch the actual value of the variable the bar1d controls and will show the correct value. You must call this method, if you change the value of a variable, which is also controlled by a bar1d.

9.1.3.2 `Button`

(subclass of 9.1.3.14 `Interactive`)

A button is a simple object, which will send its `action method` to itself if you click on it. You can change this `action method` using the method `"set-action"` (see 9.1.3.14). Normally this will be the `"default-action"` (9.1.3.14), which will send the `method` you can choose with `"set-method"` (9.1.3.14) to the `instance` you set using `"set-instance"` (9.1.3.14). Also the `"default-action"` will switch the button again. If you only set the `method` a button sends, and you set no `instance`, the button will send the `method` to the current object of the `manager` (see section 4.3.2). You can set the `instance` and `method` also in the `"new-instance"` method of a button. If you don't know the `instance` or `method` at that time, or you want to set a different `action method`, you can give a `NULL` parameter as the `method` and/or as the `instance`.

GRAPE(button, "default-action")()

> **BUTTON *button**

The default action of a `Button`. Sends the Button's method to the according instance. If instance is `NULL`, the method is sent to the current object of the manager.

GRAPE(Button, "new-instance")(meth, inst, name)

> **CLASS *Button**
> **char *meth**
> **INSTANCE *inst**
> **char *name**

Creates and returns a new Button instance named `name`. When pressed a Button calls its `action`, the preset action `"default-action"` will send the method `meth` to the instance `inst`; for other purposes another action can be set using `""set-action"`.

GRAPE(button, "pop-items-layer")()

> **BUTTON *button**

Action to be set if the Button is used to show the layer its `instance` resides in. If the layer is already added to the manager, it is simply brougth into front, else it is added to the manager.

GRAPE(button, "pop-layer-action")()

BUTTON *button

Action to be set if the Button is used to create a Layer. When the Button is pressed, it checks if the Layer already exists. If that is true, the Layer is made visible and popped to be the upper one. Else the `method` is called on the `instance`, which has to create the Layer and to insert it into the layer-list of the Button using `"layer-list-add"`; to make this easy the Button is passed as argument of the `method`.

GRAPE(button, "set-state")(on_off)

BUTTON *button
int on_off

If `on_off == ON`, the Button is drawn pressed, if `on_off == OFF` it is unpressed.

GRAPE(button, "switch-state")()

BUTTON *button

Toggle the pressed state of a `button`.

9.1.3.3 CheckField

(subclass of 9.1.3.12 `Group`)

A checkfield is the combination of several checkboxes, together with a variable dependence structure. There are some predefined easy dependencies like a radio-button dependence, where only one button at the same time can be pressed. It is also possible to define complex own dependencies. The single checkboxes alternative can switch single bits of one variable of the type CHECKFIELD_VAR (at least 32 bits) or each checkbox has got its own variable, which is switched on and off. Combinations of this variable strategies for one checkfield are possible. Each checkbox in a checkfield is referenced by a unique bit in a CHECKFIELD_VAR. This field will be called the mask of this checkbox in the following context. If the checkfield has one variable whose bits are switched, each checkbox will switch its reference bit. For the dependencies each checkbox of a checkfield has got four dependency bit-masks. Two of them are used, if the checkbox is switched on, we will call them check-masks, the other ones are used if it is switched off and we will call them unset-masks. If you switch the checkbox, it will first switch off all checkboxes of the checkfield, whose bits are not set in the (un-)check-AND-mask. After that all checkboxes, whose bits in the (un-)check-XOR-mask are set, are switched. You can use predefined masks if you want simple a checkfield like one with a radio-button behavior. Otherwise you can set the mask using the method `"set-dependency-mask"` or `"set-checkfield-bits"`. Pay attention that some methods needed for using `checkfield`s are methods of the class `checkbox`. Look there to get all information.

(CHECKFIELD_VAR *) GRAPE(checkfield, "get-bitfield")()

CHECKFIELD *checkfield

Returns the sum bit mask containing all pressed Checkboxes' mask bits.

(CHECKFIELD_VAR *) GRAPE(checkfield, "get-free-mask")()

CHECKFIELD *checkfield

Returns a bit mask containing a single bit not already used by other Checkboxes of this Checkfield.

(CHECKBOX *) GRAPE(checkfield, "insert-checkbox")(checkbox, this_mask)

CHECKFIELD *checkfield
CHECKFIELD_VAR this_mask

Adds `checkbox` to a `Checkfield`, using `this_mask` as its unique identifying bitmask. See also `"new-checkbox"`.

(CHECKBOX *) GRAPE(checkfield, "new-checkbox")(name, this_mask, var)

> **CHECKFIELD *checkfield**
> **char *name**
> **CHECKFIELD_VAR this_mask**
> **CHECKFIELD_VAR *var**

Creates a new Checkbox named `name` inside the Checkfield. This checkbox will be referenced by the checkfield only with the bit-mask `this_mask`. So it is necessary to set one unique bit in this bitmask. If have have chosen an automatic mask-mode for the checkfield, the dependency bitmasks of the other checkboxes will be set correctly. If `var != NULL` the checkbox will control its own value (which will be set to `0` or to `1`), instead of the bit in the variable of the checkfield. Most times the checkfield looks better if you set a size for the single checkboxes after creating them. Because the Checkbox is already inserted into the Checkfield when you can call `"set-pref-size"` you have to call the method "calc-inter" on the checkfield after all checkboxes are added and correctly resized.

GRAPE(CheckField, "new-instance")(name, var)

> **CLASS *CheckField**
> **char *name**
> **CHECKFIELD_VAR *var**

`var` is the pointer to the variable where the state of the single checkboxes is referenced by their bits. This pointer can be a NULL pointer if you want to use a single state variable for each checkbox. If you use this variable, it should have a state which is compatible with the dependency-masks, to have a correct setting at startup.

GRAPE(checkfield, "set-dependency-masks")(check_and, check_xor, uncheck_and, uncheck_xor)

> **CHECKFIELD *checkfield**
> **CHECKFIELD_VAR check_and, check_xor**
> **CHECKFIELD_VAR uncheck_and, uncheck_xor**

This method sets the default connection bitmasks which will used for each new checkbox created using `"new-checkbox"`, if the active mask-mode is `G_CHECKFIELD_CUSTOM`. This provides an easy interface to create multiple Checkboxes with the same behaviour.

To be used for more than one special Checkbox the values are rotated until the least significant bits of the set masks are at the actual bit position.

GRAPE(checkfield, "set-variable")(variable)

> **CHECKFIELD *checkfield**
> **CHECKFIELD_VAR *variable**

Set the variable controlled by the `CheckField`, i.e. the bitmask whose bits are switched by `Checkboxes` not having an own variable. This allows to change the setting coming from the `"new-instance"` argument.

GRAPE(checkfield, "set-mask-mode")(mask_mode)

> **CHECKFIELD *checkfield**
> **int mask_mode**

Set a global connection mode for the Checkboxes. This must be done before the first `"new-checkbox"`.
`G_CHECKFIELD_RADIO`: Radio-buttons, one and only one can be `ON`.
`G_CHECKFIELD_CBOXES`: Checkboxes, arbitrary combinations allowed.
`G_CHECKFIELD_CUSTOM`: Use special user defined masks (`"set-dependency-masks"`) for each Checkbox.

9.1.3.4 `Checkbox`

(subclass of 9.1.3.2 `Button`)

A Checkbox toggles a flag. Checkboxes can be combined to a CheckField which allows defining connection rules like radio-button-mode.

GRAPE(checkbox, "get-flag")(on_off)

CHECKBOX *checkbox
int *on_off

Get the associated flag. `*on_off` is set `ON` resp. `OFF` depending on the state.

GRAPE(Checkbox, "new-instance")(name, value)

CLASS *checkbox
char *name
int *value

A checkbox can be used inside of a checkfield, or stand alone. If you want to use a stand alone checkbox the pointer `value` must not be a NULL pointer.

GRAPE(checkbox, "set-checkfield-bits")(that, bit)

CHECKBOX *checkbox
CHECKBOX *that
int bit

Using this method the dependency with one other Checkbox (`that`) can be set.

`bit` contains the combined values to be set in the 4 masks of `checkbox`, to set a `1` use the appropriate constant,
`G_CHECKFIELD_CHECK_AND`,
`G_CHECKFIELD_CHECK_XOR`,
`G_CHECKFIELD_UNCHECK_AND`,
`G_CHECKFIELD_UNCHECK_XOR`.

GRAPE(checkbox, "set-dependency-masks")(check_and, check_xor, uncheck_and, uncheck_xor)

CHECKBOX *checkbox
CHECKFIELD_VAR check_and, check_xor
CHECKFIELD_VAR uncheck_and, uncheck_xor

Directly set the dependency mask of one Checkbox.

GRAPE(checkbox, "set-flag")(on_off)

CHECKBOX *checkbox
int on_off

Set the associated flag.

GRAPE(checkbox, "set-mask")(mask)

CHECKBOX *checkbox
CHECKFIELD_VAR mask

Set a new unique bit mask for the Checkbox.

GRAPE(checkbox, "set-variable")(var)

CHECKBOX *checkbox
int *var

Set `var` as new variable pointer of the `Checkbox`, i.e. change the value passed as argument of `"new-instance"` or even `"new-chekcbox"` of class `CheckField`.

If `checkbox` is contained in a `CheckField` setting the variable to `NULL` makes `checkbox` use the field's variable. Vice versa if `checkbox` is part of and using the variable of a `CheckField` its gets an independent variable if non `NULL` is set.

GRAPE(checkbox, "switch-flag")()

CHECKBOX *checkbox

Toggle the flag.

GRAPE(checkbox, "switch-off")()

CHECKBOX *checkbox

Switches the Checkbox flag off. Since it has no arguments this method can be used by Buttons.

GRAPE(checkbox, "switch-on")()

CHECKBOX *checkbox

Switches the Checkbox flag off. Since it has no arguments this method can be used by Buttons.

9.1.3.5 `Color_Sel`

(subclass of 9.1.3.12 `Group`)

A Color_Sel is designed to implement an easy way to control color-vectors. The init-parameter is a `VEC3`. The Color_Sel will show this color-vector and enables the editing of it using the RGB- or the HSV-model. The color-vector will always contain the RGB-values. As for all interactives the Color_Sel will call its action method if the values of the color-vector are changed. The default action will send the `method` to the `instance`.

(char *) GRAPE(color_sel, "get-color-model")()

COLOR_SEL *color_sel

Return the constant string `"RGB"` or `"HSV"` depending on the currently active color model.

GRAPE(Color_Sel, "new-instance")(name, rgb)

CLASS *Color_Sel
char *name
VEC3 rgb

Create a new Color_Sel instance to edit `rgb`.

GRAPE(color_sel, "set-color-model")(model)

COLOR_SEL *color_sel
char *model

Change the color-model for the Color_Sel to RGB or HSV depending on `model`, which has to be `"RGB"` or `"HSV"`.

GRAPE(color_sel, "set-var")(var)

COLOR_SEL *color_sel
VEC3 var

Set a new variable, `var`, to be edited with `color_sel`. If `var` equals `NULL`, `color_sel` is switched `INACTIVE`, else `ACTIVE`.

GRAPE(color_sel, "switch-color-model")()

COLOR_SEL *color_sel

Change the color-model for the Color_Sel between RGB and HSV. Used by the RGB/HSV-button inside the Color_Sels; might be useful to be called by hand if HSV editing mode is to be used.

9.1.3.6 `ComboButton`

(subclass of 9.1.3.7 `Cycle_Button`)

A Cycle_Button that does not simply change it value when clicked but opens a list presenting all possible values. Select an entry by releasing the left mouse button.

9.1.3.7 `Cycle_Button`

(subclass of 9.1.3.2 `Button`)

A Button with a number of different labels, each possessing its own definition value. This value will be given as a parameter if the button is selected.

```
typedef struct {
  int value;
  char *label;
} CYCLE\_LABEL;
```

GRAPE(c_button, "add-label")(label)

> **CYCLE_BUTTON *c_button**
> **CYCLE_LABEL *label**

Add the new label `label` add the end of `c_button`'s label list.

GRAPE(c_button, "get-value")(value)

> **CYCLE_BUTTON *c_button**
> **int *value**

Returns the actual `value` of the Cycle_Button.

GRAPE(Cycle_Button, "new-instance")(method, inst, name, label_list)

> **CLASS *Cycle_Button**
> **char *method**
> **INSTANCE *inst**
> **char *name**
> **CYCLE_LABEL *label_list**

Creates a Cycle_Button. `label_list` is an array of CYCLE_LABELs, terminated with a `NULL` entry. If the Cycle_Button is selected, it will send the method `method` to the instance `inst`, using the new value of the Cycle_Button as a parameter.

GRAPE(c_button, "remove-label")(value)

> **CYCLE_BUTTON *c_button**
> **int value**

Remove the label having value `value` from the label list.

GRAPE(c_button, "set-value")(value)

> **CYCLE_BUTTON *c_button**
> **int value**

Set the value of the Cycle_Button to `value`. The `label` of the button will be updated to show the correct label.

9.1.3.8 `FileList`

(subclass of 9.1.3.30 `XList`)

In contrast to the `Xlist` a FileList is filled with data automatically. The only information needed are the path, a wildcard for the file names and the open mode, which specifies if the files need to be readable, writable or both. If the user has finished his selection, the FileList will call a method on an instance, both defined from the programmer. This method gets a pointer to the FileList as a parameter. Using this parameter, for example named `fl`, the called method can determine which item was chosen by the user. The selection can be found in `fl->chosen`. If the user canceled the selection, using the cancel-button, `fl->chosen` will be equal to `NULL`. If the FileList is implemented as an overlay menu in its own layer, the helper function `g_hide_layer` will automatically hide the layer of the FileList. A FileList which is created using XList_Button–`"new-for-filelist"` has got its own layer, so the above helper functions can be used.

```
typedef enum open_mode {
  omRead,
  omWrite,
  omWriteRead
} OPEN_MODE;
```

Example: a Button which pops up a FileList to read data of instance `inst`.

```
INST *inst_file_selected (FILELIST *filelist)
{
  ...
  if (filelist->chosen) {
    ... /* Read data from the standard TextDevice */
  }
  else {
    /* User pressed "Cancel" or file doesn't exist  */
  }
  g_hide_layer ((ITEM *)filelist);
  ...
}

{
  XLIST_BUTTON *button;

  button = (XLIST_BUTTON *) GRAPE (XList_Button, "new-for-filelist")
    ("file-selected", inst, "Load File", "get-files", "*.st", omRead);
}
```

GRAPE(filelist, "get-files")()

FILELIST *filelist

Default method to be used as directory reading method, to fill the FileList.

GRAPE(FileList, "new-instance")(meth, inst, name, ext, fmodus)

CLASS *FileList
char *meth
INSTANCE *inst
char *name
char *ext
OPEN_MODE fmodus

Creates a new FileList, which will show all files matching the wildcard `ext`. If a file is selected, the method `meth` will be send to the instance `inst`. The easiest way to use a FileList is to use XList_Button–`"new-for-filelist"`.

GRAPE(filelist, "share-path")(other)

FILELIST *filelist, *other

`filelist` and `other` share one path variable now. This is good for coupled read/write lists like in the manag menu.

9.1.3.9 `Function_Bar1d`

(subclass of 9.1.3.24 `Slider`)

Function_Bar1d is the abstract superclass of Function_Ruler and Function_Slider. In addition to the Bar1d the class Function_Bar1d provides methods and variables to store and edit a spline. If the effect is set `ON`, the spline is used to describe the dependency from a variable, e.g. another Ruler. The only method which is necessary in the usage of a Function_Bar1d is the method `"set-effect"`. When the effect of a Function_Bar1d is switched `ON` it will be added as a side-effect of the main time ruler.

GRAPE(function_bar1d, "set-effect")(on_off)

> **FUNCTION_BAR1D *function_bar1d**
> **int on_off**

With this method, the dependence of the Function_Bar1d can be switched `ON` or `OFF`. If the dependence is switched `ON` the value of the Function_Bar1d will be set using the spline each time the Function_Bar1d gets an `"update"`.

GRAPE(function_bar1d, "set-spline")(spline)

> **FUNCTION_BAR1D *function_bar1d**
> **SPLINE1D *spline**

Set `spline` as function for `function_bar1d`, using `ASSIGN`.

GRAPE(function_bar1d, "unset-spline")()

> **FUNCTION_BAR1D *function_bar1d**

Delete the spline of `function_bar1d`. (Cf. 5.4.4.1)

9.1.3.10 `Function_Ruler`

(subclass of 9.1.3.9 `Function_Bar1d`)

GRAPE(Function_Ruler, "new-instance")(spline, fun_arg, var, name, atype)

> **CLASS *Function_Ruler**
> **SPLINE1D *spline**
> **double *fun_arg**
> **void *var**
> **char *name**
> **DECLARATION_FORMAT atype**

Creates a new instance of the class Function_Ruler which is named `name`. The variable `var`, which is controlled by the ruler must be of the type defined by `atype`. If no `spline` is given (`spline == NULL`) the Function_Ruler will try to load a spline from a file, whose name is equal to the name of the Function_Ruler and the extension .sp (all spaces in the name are converted to underscores in the filename). If no matching file is found, the Function_Ruler will create a new linear spline. The parameter `fun_arg` is the argument of the spline function and must be of type `double`. Most time this argument is the main time or the scene time.

9.1.3.11 `Function_Slider`

(subclass of 9.1.3.10 `Function_Ruler`)

GRAPE(Function_Slider, "new-instance")((see superclass method))

> **CLASS *Function_Slider**
> **(see superclass method)**

Overloading inherited method of class `9.1.3.9 Function_Bar1d`.

9.1.3.12 `Group`

(subclass of 9.1.3.14 `Interactive`)

A group is used to create structured menus very easy. It contains several items, but it's handled like one single item from the upper groups. The items of a group are placed automatically inside the group using their preferred positions and sizes. The group can also have a border and a title to make the group structure visible.

```
typedef enum border_flags{
  bfNoBorder,
  bfBorder,
  bfTitle
} BORDER_FLAGS;
```

GRAPE(group, "add-inter")(item)

> **GROUP *group**
> **ITEM *item**

Add the item `item` to the group. If there is no place for the item in the group and all its replacement groups, the item will be added invisible.

GRAPE(group, "add-first-inter")(item)

> **GROUP *group**
> **ITEM *item**

Add the item `item` to the group. Like `"add-inter"`, but `item` is added as first and not as last entry.

GRAPE(group, "add-replacement")(new_replace)

> **GROUP *group**
> **GROUP *new_replace**

Add the group `"new_replace"` to the list of replaced groups. See 9.1.1.2 for details about replacing.

GRAPE(group, "calc-inter")()

> **GROUP *group**

Recalculates the position and size of all items. This method is used from some subclasses of group, for example XList or Spline_Editor, to adapt the size of their items. The method should be called, if the pref-size or -pos of an item has been changed, which already resides in group.

GRAPE(group, "change-border")(new_border)

> **GROUP *group**
> **BORDER_FLAGS new_border**

Switches the border flags of a group. Using that method the border and title of a group can be switched on and off. Do not use a title without a border.

GRAPE(group, "contains")(item)

> **GROUP *group**
> **ITEM *item**

Returns `NULL`, if the item is not contained in the group.

(int *) GRAPE(group, "count-entries")(class)

GROUP *group
CLASS *class

This function will return a pointer to the number of items contained in the group and having the superclass `class`. If `class == NULL` all items will be counted.

GRAPE(group, "free")()

GROUP *group

First of all this group will send `"delete"` to all items, which are contained in the group. After that the group will remove itself.

(ITEM *) GRAPE(group, "get-entry")(item, class)

GROUP *group
ITEM *item
CLASS *class

If `item == NULL`, this function will return the first item which is contained in the group and has got the superclass `class`. Otherwise the next item with right class will be returned.

GRAPE(group, "get-size")(x, y)

GROUP *group
double *x, *y

Returns the inner size of the group in `x` and `y`. This size is the width or height of the rectangular space used for Items.

GRAPE(Group, "new-instance")(name)

CLASS *Group
char *name

Create a new Group instance. This Group is empty, has no border and a very small size.

GRAPE(group, "remove-inter")(item)

GROUP *group
ITEM *item

Remove the Item `item` from the group.

GRAPE(group, "remove-replacement")(old_replace)

GROUP *group
GROUP *old_replace

Remove the Group `"old_replace"` from the list of replace Groups.

GRAPE(group, "send-add-inter-action")()

GROUP *group

The default `add_inter_action` of a `Group`: simply call the `add_inter_actions` of the contained items.

GRAPE(group, "send-to-all-entries")(meth)

GROUP *group
char *meth

Sends the method `meth` to all Items, which are contained in the group. The method `meth` must not have parameters.

GRAPE(group, "set-redraw-flag")(redraw_flag)

> **GROUP *group**
> **REDRAW_FLAG redraw_flag**

Calls `"set-redraw-flag"` on all Items, which are contained in the Group and sets the redraw flag of the Group itself as well.

GRAPE(group, "send-remove-inter-action")()

> **GROUP *group**

The default `remove_inter_action` of a `Group`: simply call the `remove_inter_actions` of the contained items.

GRAPE(group, "update")()

> **GROUP *group**

Overloading inherited method of class `9.1.3.15 Item`. Will call the `"update"` method of each interactive, which is contained in the group.

9.1.3.13 `Input`

(subclass of 9.1.3.14 `Interactive`)

An input allows to view and edit `int-`, `float-`, `double-` and `char *`-variables.

```
typedef enum declaration_format{
  dfInt,
  dfFloat,
  dfDouble,
  dfString,
  dfRegexp
} DECLARATION_FORMAT;
```

GRAPE(Input, "new-instance")(var, name, type, max_size)

> **CLASS *Input**
> **void *var**
> **char *name**
> **DECLARATION_FORMAT type**
> **int max_size**

Create a new `Input` instance: this one will edit the variable `var` of type `type`, in a `max_size`-1 characters long string. The instance name is `name`.

GRAPE(input, "set-keyactive")()

> **INPUT *Input**

Make `input` the active `Input` field, if none was set before.

GRAPE(input, "set-max-size")(max_size)

> **INPUT *Input**
> **int max_size**

Change the maximum string size of `input` to be `max_size`-1.

GRAPE(input, "set-print-format")(format)

INPUT *input
char *format

Set the `printf`–format-string, which is used from the input to show its value, if the input is not waiting for keyboard events.

When the input calls `sprintf` the label of the Input is passed as a second argument. To display name and (`int`) value of the Input you can use the format string `"%2$s: %1$d"` for example. Normally only SGI's C-library understand formats containing $, but it is emulated on other machines.

GRAPE(input, "set-tab-action")(tab_action)

INPUT *input
char *tab_action

Change the `tab action` to `tab_action`. This is the method the input will send to itself if the tab-key is pressed.

GRAPE(input, "set-type")(type)

INPUT *input
DECLARATION_FORMAT type

Set the type of controlled the variable.

GRAPE(input, "set-var-and-type")(var, type)

INPUT *input
void *var
DECLARATION_FORMAT type

Set the variable pointer to a new variable, together with setting its type.

GRAPE(input, "set-variable")(var)

INPUT *input
void *var

Set the variable pointer to a new variable. Don't forget to set the correct type of the variable.

9.1.3.14 `Interactive`

(subclass of 9.1.3.15 `Item`)

The class interactive is the abstract superclass for all items, which must react on user events like mouse actions or keyboard input.

If the interactive reacts on a user action, this means that the action is really finished, it calls its `action`-method.

As a difference to an Item every Interactive has side-effect list of all Items which depend on the Interactive in any way. All these items get an `"update"` if necessary.

```
typedef enum {
  rfOff,
  rfDisplist,
  rfOn
} REDRAW_FLAG;
```

GRAPE(interactive, "add-layers")()

INTERACTIVE *interactive

Show all layers, which are in the interactive's layer list. See also 9.1.3.14.

GRAPE(interactive, "add-side-effect")(effect)

INTERACTIVE *interactive
INTERACTIVE *effect

Add `effect` as a side-effect interactive of the interactive. All side-effect interactives get an `update` method if the `action` method is executed. This is useful to keep several interactives up to date, which all control the same variable.

GRAPE(interactive, "default-action")()

INTERACTIVE *interactive

The default `action` for all Interactives. Sends the `method` to the `instance`.

(int *) GRAPE(interactive, "is-active")()

INTERACTIVE *interactive

The method will return a value different to `NULL`, if the interactive is active. This means, that not only the interactive itself is active, but all groups and the layer the interactive is part of are active.

GRAPE(interactive, "layer-list-add")(layer)

INTERACTIVE *interactive
LAYER *layer

Add the `layer` to the layer list of an Item 9.1.3.15. Together with the methods `"add-layers"` and `"remove-layers"` as add- and remove-inter-actions the layers in the list could be dependent upon the interactive and be only visible is the item is visible.

(LAYER *) GRAPE(interactive, "layer-list-contains")(member)

INTERACTIVE *interactive
LAYER *member

Will return `member`, if this layer is contained in the layer list. Otherwise it will return `NULL`.

GRAPE(interactive, "layer-list-remove")(layer)

INTERACTIVE *interactive
LAYER *layer

Remove the `layer` from the layer list of an interactive.

GRAPE(interactive, "mark-side-effect")()

INTERACTIVE *interactive

Update all instances which are in the side-effect list of the interactive. To avoid problem with recursive calls of the method `"update"` an update method should never call `"mark-side-effect"`.

GRAPE(interactive, "remove-layers")()

INTERACTIVE *interactive

Will hide all layers, which are in the layer list of the item. If you change the add-action of an item to `"add-layers"` as the add action and `"remove-layers"` as the remove action, all layers of the layer list are always removed and added together with the interactive.

GRAPE(interactive, "remove-layer-action")()

INTERACTIVE *interactive

Remove all layers contained in the layer list of `interactive`. This is done by calling `"remove"` on them, i.e. they are deleted, not only hidden.

GRAPE(interactive, "remove-side-effect")(effect)

INTERACTIVE *interactive
INTERACTIVE *effect

Remove a side effect notifier.

GRAPE(interactive, "send-to-layers")(method)

INTERACTIVE *interactive
char *method

Send `"method"` to all entries of the layer list of `interactive`.

GRAPE(interactive, "set")(active)

INTERACTIVE *interactive
int active

Set active state. The possible values for `active` are `ACTIVE` and `active == INACTIVE`.

Inactive Interactives are grayed out to prevent usage.

When a Group is inactive all contained Items will behave inactive even if their active flag is set to active.

GRAPE(interactive, "set-action")(name)

INTERACTIVE *interactive
char *name

Set the action method to of an `name` interactive. This action will be sent to the item, if the user has touched it, for example pressed a button or finished input typein. Pay attention, that a Function_Bar1d could change its value without being changed by the user. To handle this the Function_Bar1d will send its `value-changed`–method, which is called every time the value of its variable has changed.

GRAPE(interactive, "set-instance")(inst)

INTERACTIVE *interactive
INSTANCE *inst

Set the `instance`. This is the instance the action method sends the method to.

GRAPE(interactive, "set-label")(label)

INTERACTIVE *interactive
char *label

Set the label of `interactive`, i.e. the string that is drawn. An example for a label is the text on a `Button` or the title of a `Group`. Do not mix up the label and the instance name of an `Interactive`.

GRAPE(interactive, "set-method")(name)

INTERACTIVE *interactive
char *name

Set the method, which should be sent from the action method, to `method`.

GRAPE(interactive, "set-redraw-flag")(redraw_flag)

INTERACTIVE *interactive
REDRAW_FLAG redraw_flag

Change the kind of graphic redraw an item generates when used. This function is used for item which shouldn't generate a graphic redraw, for example a scrollbar in a list. `Interactive`s inherit the redraw flag of containing `Group`s.

GRAPE(interactive, "spread-redraw-flag")(redraw_flag)

INTERACTIVE *interactive
REDRAW_FLAG redraw_flag

Set the redraw flag of `interactive`. If `interactive` is a `Group`, set the redraw flag of all contained `Interactivess` as well.

GRAPE(interactive, "switch")()

INTERACTIVE *interactive

Toggle between active and inactive state. Inactive Interactives are grayed out to prevent usage.

9.1.3.15 `Item`

(subclass of Root)

Abstract superclass of all visible and interactive objects in the control window. Implements basic functionality for redraw management, positioning and sizing.

Every Item can have a list of associated Layers, one possible application is to hide a Layer when the Item is invisible, to reduce the number of layers. See 9.1.3.14 Interactive for the methods.

(GROUP *) GRAPE(item, "get-group")()

ITEM *item

Returns the Group which contains the Item. If the Item has not been added to a group the method will return `NULL`.

(char *) GRAPE(item, "get-item-path")()

ITEM *item

Return the path through `Layers` an `Groups` to `item`. The return value is allocated via `g_strdup` so you have to free it after use, e.g. using `g_strfree`.

One example for an item path is the exit `Button`: `"/main/top-row/exit"`

(ITEM *) GRAPE(item, "is-visible")()

ITEM *item

Will return a value unequal to `NULL`, if the item is not part of the manager at that moment. An item is also called visible if it is partial or totally hidden behind a layer.

GRAPE(Item, "new-instance")(name)

CLASS *Item
char *name

Creates a new item.

(ITEM *) GRAPE(item, "path-to-item")(path)

ITEM *item
char *path

Return the first `Item` that matches the path `path`. This is trivial for single `Items`, normally one sends this method to a `Group` or to the manager, which tries to find the `Item` in all of its layers.

GRAPE(item, "print-item-path-send")()

Item *item

Print the item path of `item`. Designed for use by one's `.graperc`.

GRAPE(item, "set-add-action")(add, remove)

ITEM *item
char *add, remove

Change the method the item will call when its visibility-state is changed, which means that it has been added to or removed from the manager. If the item becomes visible it will call the method `add`, if it becomes invisible it will call `remove`. This mechanism is used to add and remove item dependent layers.See also 9.1.3.14

GRAPE(item, "set-color")(rgb)

> **ITEM *item**
> **VEC3 rgb**

Changes the backgroundcolor of an item to a special RGB-value. You should use this method only if it is really necessary, because the colors in the X11-device are stored as indexed-colors and there are only few color indexes available. You should better use `"set-color-index"` to use a default color index.

GRAPE(item, "set-color-index")(color_index)

> **ITEM *item**
> **int color_index**

Change the background color of an item to the color with the index `color_index`. There are predefined index constants like `RED_COLOR`, `LINE_COLOR`, `BACKGROUND_COLOR` a.o.

GRAPE(item, "set-fill-mode")(mode)

> **ITEM *item**
> **int mode**

Changes the mode for the autopositioning of the item. If `mode == MENU_FILL_TOP` the item will try to find a place as near to the upper left corner of the group as possible. If `MENU_FILL_BOTTOM` it will try to find a place as near to the lower right corner of the group as possible

GRAPE(item, "set-help-text")(help_text)

> **ITEM *item**
> **char *help_text**

Set the help text of `item`. This is the text that is displayed when you click Control-MiddleMouseButton on it.

GRAPE(item, "set-label")(label)

> **ITEM *item**
> **char *label**

Set the label of an Item. The label is the displayed text.

GRAPE(item, "set-pref-pos")(px, py)

> **ITEM *item**
> **double px, py**

Set the preferred position of an item to `px` and `py`. The position is the position of the left lower corner of the item. Alternative you can use the magical const `MENU_AUTO_POSITION` to position the item automatically. See also 9.1.1.2.

GRAPE(item, "set-pref-size")(sx, sy)

> **ITEM *item**
> **double sx, sy**

Set the preferred size of an item to `sx` and `sy`. Alternative you can use the magical const `MENU_MAX_SIZE`. If the item doesn't fit in the group (and all possible replacement groups) it will be added, but is invisible. If you want to change the size of an item, which is still added to a group, you need to send the method `"calc-inter"` to the group. If you don't know the group the method `"get-group"` will return it. See also 9.1.1.2.

GRAPE(item, "set-refresh")(on_off)

ITEM *item
int on_off

`on_off == ON`: draw the Item the next time the Manager checks its refresh state.

GRAPE(item, "set-rel-size-mode")(on_off, on_off)

ITEM *item
int on_off

GRAPE(item, "update")()

ITEM *item

Update the internal state. If a pointed to variable was changed by hand `"update"` tells the Item to update its appearance.

9.1.3.16 Layer

(subclass of 9.1.3.12 `Group`)

A movable overlay Group.

A layer must be added to the manager using the method `"add-layer"` to add it to the manager. To remove it from the manager use the function `"remove-layer"`. If the layer depends upon an item, it may be removed or added automatically using this methods. It is also possible to add display method dependent layer using it as a each other normal item.

This example generates a layer, which contains a checkbox, a ruler and a button, which will hide and delete the layer.

```
LAYER *layer;
MANAGER *mgr;
BUTTON *butt;
CHECKBOX *checkbox;
RULER *ruler;
double sizex, sizey;
int my_flag;
static double my_double;


/* Get the standard Manager. If none existed it will be created.*/
mgr = (MANAGER *)GRAPE (Manager, "get-stdmgr")();

/* Create a new layer. */
layer = (LAYER *)GRAPE (Layer, "new-instance")("my third layer");

/* Set size of layer */
GRAPE (layer, "set-pref-size")(10.0, 10.0);

/* Add it to the manager. It will be drawn at the next Redraw. */
GRAPE (mgr, "add-layer")(layer);

/* Get the layer's inner size, e.g. without the border.*/
GRAPE (layer, "get-size")(&sizex, &sizey);

/* Create a dismiss button with removes the layer by sending "remove" if pressed.*/
butt = (BUTTON *) GRAPE (Button, "new-instance")("remove", layer, "dismiss");
GRAPE (butt, "set-pref-size")(sizex / 2, 1.0);

/* Button will be positioned at the very bottom of the layer. */
```

```
GRAPE (butt, "set-fill-mode")(MENU_FILL_BOTTOM);

/* Put button into layer. */
GRAPE (layer, "add-inter")(butt);

/* Create a new ruler which points to a double Variable. */
ruler = (RULER *) GRAPE (Ruler, "new-instance")
        (&my_double, "my ruler", dfDouble);

/* The ruler will fill the layer even if it is resized. */
GRAPE (ruler, "set-pref-size")(MENU_MAX_SIZE, 1.0);
GRAPE (ruler, "set-fill-mode")(MENU_FILL_TOP);
GRAPE (layer, "add-inter")(ruler);
checkbox = (CHECKBOX *) GRAPE (Checkbox, "new-instance")("my checkbox", &my_flag);
GRAPE (checkbox, "set-fill-mode")(MENU_FILL_TOP);
GRAPE (layer, "add-inter")(checkbox);
```

GRAPE(layer, "fix-size")(on_off_x, ob_off_y)

LAYER *layer
int on_off_x, ob_off_y

Fix the size of `layer` in the given directions. This only prohibits rescaling `layer` with the mouse, methods are not affected. Since the size of a `Layer` should be the user's responsibility the use of this method is not encouraged.

GRAPE(layer, "hide")()

LAYER *layer

Hides a layer. If the layer depends on an item, it will be removed from the items layer list. The invert method is `show`

(LAYER *) GRAPE(layer, "is-visible")()

LAYER *layer

GRAPE(Layer, "new-instance")(name)

CLASS *Layer
char *name

Create a new layer.

GRAPE(layer, "remove")()

LAYER *layer

Will hide a layer first and then delete it.

GRAPE(layer, "set-fill-mode")(mode)

LAYER *layer
int mode

Overloading inherited method of class `9.1.3.15 Item`. Additional a layer could have the fill-mode `MENU_MOUSE_POS`. Using this mode the layer will pop-up relatively to the actual mouse position, if the layer is added to the manager.

GRAPE(layer, "set-pref-pos")(px, py)

LAYER *layer
double px, py

Overloading inherited method of class `9.1.3.15 item`. This is the position of the lower left corner of the layer in absolute coordinates of the ControlDevice.

GRAPE(layer, "show")()

LAYER *layer

Will add the `layer` to the manager and add it, if necessary, to the dependency list of the item, the layer depends on again. This method is the inverse method to `hide`.

9.1.3.17 `Manager`

(subclass of Root)

The manager is the connection between the user's actions, the mathematical objects, the hardware drivers and the graphical view. A longer description can be found in section 9.1.1.1 and in section 4.

The manager has got a new concept for long numerical processes, which will show the progress and enable the user to stop or abort the numerics. To do so, you have to create a wait layer, as it is shown in the following example:

An example wait layer:

```
void numerics (void)
{
  MANAGER *mgr;
  int i;

  mgr = (MANAGER *) GRAPE (Manager, "get-stdmgr")();

  /* inactivate everything, open a layer containing a percent bar
   * which shown the current state of i, 0 on the left side, 1000
   * on the right side */

  GRAPE (mgr, "disable-menu")(&i, 0.0, 1000.0, dfInt);
  for (i = 0; i < 1000; i++) {
    ...; /* numerics */

    /* Let the Manager process the pending events, allow the user
     * to press "pause" to pause, update the percent bar.
     * if "abort" is pressed we have to terminate the loop
     */
    if (!GRAPE (mgr, "process-events")(NULL))
      break;
  }
  GRAPE (mgr, "enable-menu")();
}
```

`void g_ctl_draw_procedure (MANAGER *mgr);` Redraw the standard GraphicDevice. That is: draw the current object.

GRAPE(manager, "add-cross-depend")(instance, method)

MANAGER *manager
INSTANCE *instance
char *method

Every time `"recalc-cross"` is called in `manager` it will call the following: `GRAPE (instance, method)(current_object)`.

GRAPE(manager, "add-inter")(item)

MANAGER *manager
ITEM *item

Tries to add the `item` to the first opt-menu. This method will do the same as `GRAPE(manager, "add-inter-for-opt")(item, 0)`

GRAPE(manager, "add-inter-for-opt")(item, nr)

> **MANAGER *manager**
> **ITEM *item**
> **int nr**

Tries to add the `item` to the option-menu `nr`. If the item doesn't fit in this opt-menu it will be automatically put into the next one, which has enough place for it (see also 9.1.1.2). If there is no space left for this item in any opt-menu, NULL will be returned.

GRAPE(manager, "add-layer")(layer)

> **MANAGER *manager**
> **LAYER *layer**

Add the `layer` as the new top-layer to the manager.

GRAPE(manager, "check-refresh")()

> **MANAGER *manager**

Redraw all `Item`s which redraw flag in `ON` and reset the redraw flag.

GRAPE(manager, "delete-item")(item)

> **MANAGER *manager ITEM *item**

Add the `item` to a list of items, which will be removed at the next possible time. This method is the only way to delete an item which is still in use. For example: A Button wants to delete the Layer is is part of, if you press the button. If the Button sends in its action method a "delete" method to the layer, the layer and all its items will be deleted at that moment. After that GRAPE will try to continue the action- and the cause-action method of the button, but the button is deleted. Doing it this way GRAPE will crash when the next method call for button is executed. So the only way is to add the layer to the delete list of the manager and the manager will delete it, being sure, that no more method calls for these objects are still in process.

(LAYER *) GRAPE(manager, "disable-menu")(value, min_value, max_value, vtype)

> **MANAGER *manager**
> **void *value**
> **double min_value, max_value**
> **DECLARATION_FORMAT vtype**

Disables the whole Manager and adds a new Layer, which only contains a "abort" and a "pause" Button and a Slider, which shows the actual value of `value`. The type of the variable should be selected using the `vtype` parameter. The Slider will be scaled using the `min_value` and the `max_value`.

This method initializes the support of long numeric processes, which is described with an example at the top of this section. After the call of this method the numeric process has to call the method `"process-events"` periodically to enable the usage of the Button and to update the Slider, which should show the progress of the numerics. After the numeric is finished or aborted, the method `"enable-menu"` must be called to delete the layer and enable the manager again. The usage of this mechanism can be used recursively to enable depending numeric processes.

GRAPE(manager, "enable-all-layers")()

> **MANAGER *manager**

This will enable all layers, which were disabled by a call to `"disable-menu"`. Calling this method will enable you to use the whole manager, while the numeric is in progress. This is very dangerous, because it is possible to remove methods, which are necessary for the numeric, while the numeric is still in progress.

GRAPE(manager, "enable-layer")(name)

MANAGER *manager
char *name

Enables the layer with the name `name` after he has be disabled by a call of `"disable-menu"`. This is useful to enable some special layers, while the numeric is in progress.

GRAPE(manager, "enable-menu")()

MANAGER *manager

Restore the general manager activity state a before the last `"disable-menu"`.

(INSTANCE *) GRAPE(manager, "get-current-object")()

MANAGER *manager

Returns the current object.

GRAPE(manager, "get-display-method")(display_method)

MANAGER *manager
char **display_method

Replace the string pointed to by `display_method` by a `g_strdup` copy of the currently chosen display method.

(INSTANCE *) GRAPE(manager, "get-mark-instance")()

MANAGER *manager

Return the marked instance, i.e. the instance marked in the manag-menu. This instance can change without notice, so be careful not to assume its validity across method calls.

GRAPE(Manager, "get-stdmgr")()

CLASS *Manager

Returns the standard Manager. If there is no standard manager, it will be created together with a Graphicdevice and a ControlDevice.

(SCENE *) GRAPE(mgr, "goto-instance")(object)

MANAGER *mgr
INSTANCE *object

Find an go the shortest "click"-path between the current object and `object` in the scene tree.

GRAPE(manager, "handle")(inst)

MANAGER *manager
INSTANCE *inst

Set `inst` as the current object, enter the interactive mode.

(SCENE *) GRAPE(mgr, "insert-object")(object, bury, at_head, skip_first)

MANAGER *mgr
INSTANCE *object
int bury, at_head, skip_first

Insert `object` into the scene tree in a legal way.

The first (or second, if `skip_first` is set) `Scene` above or equal the current instance is found. If this does not exist or `at_head` is set, a `Scene` with the whole scene tree as `object` is created.

After that `object` is passed to the `Scene` using `"insert-object"`, which `ASSIGN`s `object` as scene->`object` to itself or one of its `next_scene`s, if an empty one is found. If `bury` is set, the `next_scene` chain is searched to the end, otherwise a new `Scene` as direct `next_scene` is created. The return value is the `Scene`, `object` has been assigned to.

GRAPE(manager, "new-handle")(f, count)

> **MANAGER *manager**
> **void (*f)()**
> **int count**

This method checks if the manager has got `count` items registered as dependent upon the display-method `f`. If the items are already registered and added, it will return NULL. If the items are not registered, they MUST be added, using `GRAPE (mgr, "add-inter")()` or `GRAPE (mgr, "add-inter-for-opt")()`, directly after the call of `"new-handle"`. This method may only be used inside a display-method or inside a function, which is called directly from a display method. The usage outside a display method will cause an error message.

This method is used to add display dependent interactives, which are removed automatically when the display method isn't used any more.

(ITEM *) GRAPE(manager, "path-to-item")(path)

> **MANAGER *manager**
> **char *path**

Return the first `Item` matching `path`. See the method `"path-to-item"` on class `Item`.

GRAPE(manager, "pop-instance")()

> **MANAGER *manager**

Restore the previous current object after `"push-instance"`.

GRAPE(manager, "pop-layer")(layer)

> **MANAGER *manager**
> **LAYER *layer**

Pop `layer` to top position.

GRAPE(manager, "pop-message")()

> **MANAGER *manager**

Restore the main message state after a `"push-message"`.

GRAPE(manager, "process-events")(pause_flag)

> **MANAGER *manager**
> **int *pause_flag**

Process pending events. Part of the wait layer mechanism. If "abort" is pressed this method returns `NULL`. If "pause" is pressed the method will return if the pause state is released.

Always pass `NULL` as `pause_flag`.

GRAPE(manager, "push-instance")(inst, type)

> **MANAGER *manager**
> **INSTANCE *inst**
> **char type**

Set `inst` as the new current object. Valid values for `type` are

```
#define G_CROSS_EMPTY   ' '  /* no instance */
#define G_CROSS_PUSHED  'p'  /* instance is pushed, pop it to go there */
```

```
#define G_CROSS_DOWN    'd'  /* go down: current instance has to move left */
#define G_CROSS_NEXT    'n'  /* like next_scene: curr inst has to move up */
#define G_CROSS_CHAINED 'c'  /* next element in a doubly linked list,
                              * like timesteps. since this instance will
                              * show curr inst in its follow, don't push
                              * current instance */
```

. This method is called to set the instance when a button in the manag-menu is clicked. `type` is the value that was set in the `"follow"` method of the (previous) current object.

Normally you will use `G_CROSS_DOWN` for sub-instances.

An follow method example can be found in section 5.4.6.

GRAPE(manager, "push-message")(line1, line2, line3)

MANAGER *manager
char *line1, *line2, *line3

Set a three line message. The previous state of the message field is saved to be restored via `"pop-message"`.

GRAPE(manager, "recalc-cross")()

MANAGER *manager

Recalculate the viewable part of the scene tree. After calling `"follow"` on the current object the cross buttons are correctly labeled and cross dependent objects are updated.

GRAPE(manager, "remove-cross-depend")(instance, method)

MANAGER *manager
INSTANCE *instance
char *method

Remove a dependency added with `"add-cross-depend"`. If `method` is passed as `NULL` every method added with instance `instance` is removed.

GRAPE(manager, "remove-inter")(item)

MANAGER *manager
ITEM *item

Remove `item` from the containing option menu. If `item` is a layer it will be hidden by calling `"remove-layer"`.

GRAPE(manager, "remove-layer")(layer)

MANAGER *manager
LAYER *layer

Hide `layer` by removing it from the Manager's list of active layers.

GRAPE(manager, "remove-message")()

MANAGER *manager

Clear the main message. Better use `"push-message"` and `"pop-message"`.

GRAPE(manager, "send-to-layers")(method)

MANAGER *manager
char *method

Send a send-method (an arbitrary method without parameters) to all active Layers.

GRAPE(manager, "send-display-method")(method)

MANAGER *manager
char *method

Set the display method to `method`.

GRAPE(manager, "set-graphic-redraw")(new_status)

> **MANAGER *manager**
> **REDRAW_FLAG new_status**

Set the kind of graphic redraw to be done. If an object changes that is seen in the graphic window someone, e. g. the pressed button, has to set `rfOn`.

GRAPE(manager, "set-message")(str0, str1, str2)

> **MANAGER *manager**
> **char *str0, *str1, *str2**

Set a three line message, the old message is discarded. Better use `"push-message"` and `"pop-message"`.

GRAPE(manager, "set-standard-message")()

> **MANAGER *manager**

Set the standard message you see at startup.

GRAPE(manager, "set-stdmgr")()

> **MANAGER *manager**

Set `manager` as standard manager. It will be returned by `"get-stdmgr"`.

GRAPE(manager, "update")()

> **MANAGER *manager**

`"update"` the main menu.

9.1.3.18 `Matrix_Edit`

(subclass of 9.1.3.12 `Group`)

A Matrix-Edit is a Group which contains all necessary elements to change $4x4$ translation matrices. It it used for all kinds of rotations and also to change positions, like position light sources.

This class is very new and its method is not documented up to now.

9.1.3.19 `MethLayer`

(subclass of 9.1.3.16 `Layer`)

This class is very new and its method is not documented up to now.

9.1.3.20 `MethList`

(subclass of 9.1.3.30 `XList`)

A subclass of list, which enables the selection of a method of a specific class. It is possible to show only methods with a special name suffix, for example "-send"; breaking this rule you can add a some special methoda. It is possible to switch to the methods of the superclass like changing the folders of a filesystem.

GRAPE(methlist, "get-methods")()

> **METHLIST *methlist**

This method is used to fill the list with method names. It scans the GRAPE method tree and adds all method, which have the right method name suffix, to the list using `"add-to-list"`

GRAPE(MethList, "new-instance")(meth, inst, name, wanted_end, no_of_additionals, additionals)

> **CLASS *MethList**
> **char *meth**
> **INSTANCE *inst**
> **char *name**
> **char *wanted_end**
> **int no_of_additionals**
> **char **additionals**

Additionals are string which are added to the list additional to the ones found in the method tree.

9.1.3.21 `Rectangle`

(subclass of 9.1.3.15 `Item`)

A monochrome rectangle with 3D-looking border. There are no additional methods, see Item.

9.1.3.22 `Ruler`

(subclass of 9.1.3.1 `Bar1d`)

A Bar1d with a triangular moving handle.

GRAPE(Ruler, "new-instance")(var, name, atype)

> **CLASS *Ruler**
> **void *var**
> **char *name**
> **DECLARATION_FORMAT atype**

Create a ruler instance named and labeled as `name`. It will control the variable `*var`, the type of `var` must be passed in `atype`.

9.1.3.23 `Scrollbar`

(subclass of 9.1.3.1 `Bar1d`)

A simple Scrollbar. You can easily adjust the handle and bar size ratio.

GRAPE(Scrollbar, "new-instance")(var, name, atype)

> **CLASS *Scrollbar**
> **void *var**
> **char *name**
> **DECLARATION_FORMAT atype**

Create a new Scrollbar instance. See Ruler for the parameters.

GRAPE(scrollbar, "set-bar-size")(bar_size)

> **SCROLLBAR *scrollbar**
> **double bar_size**

Set the relative size of the moving part of a Scrollbar in units of `total_size`.

GRAPE(scrollbar, "set-total-size")(total_size)

> **SCROLLBAR *scrollbar**
> **double total_size**

Set `total_size` as total size in units of `*var`.

9.1.3.24 `Slider`

(subclass of 9.1.3.22 `Ruler`)

GRAPE(Slider, "new-instance")(var, name, type)

> **CLASS *Slider**
> **void *var**
> **char *name**
> **DECLARATION_FORMAT type**

Create a new Slider instance. See Ruler for the parameters.

It will be bounded to the $[0, 1]$ interval by default.

9.1.3.25 `Spline1d`

(subclass of Root)

A spline of arbitrary degree. If the degree is 3 it can be edited using Spline_Editors.

GRAPE(spline1d, "assign")(source)

> **SPLINE1D *spline1d**
> **SPLINE1D *source**

An instance method, which will assign the values of an existing spline to another. It will also adapt the length of the of the allocated lists, using `"list-free"` and `"list-alloc"`.

GRAPE(spline1d, "delete-point")(upos, fpos, eps)

> **SPLINE1D *spline1d**
> **double upos, fpos**

Delete a point from `spline1d`, with the maximal distance `eps`.

(double *) GRAPE(spline1d, "eval")(arg)

> **SPLINE1D *spline1d**
> **double arg**

Calculates the return value of the spline at the point `arg`. If the argument is out of the definition interval the function will return the value of the next reachable point (value of last point or value of the first point).

GRAPE(spline1d, "get-min-max")(min, max)

> **SPLINE1D *spline1d**
> **double *min, *max**

Calculates the minimal and maximal return values of the spline. Up to now this function simply calculates the values at `20*num_of_intervals` points of the definition interval.

(SPLINE1D *) GRAPE(spline1d, "hardcopy")()

> **SPLINE1D *spline1d**

Overloading inherited method of class `Root`.

GRAPE(spline1d, "insert-point")(upos, fpos)

> **SPLINE1D *spline1d**
> **double upos, fpos**

Insert a new point into `spline1d`. The derivative is initialized with the previous value.

(double *) GRAPE(spline1d, "inv-eval")(arg)

> **SPLINE1D *spline1d**
> **double arg**

Returns inverse of `arg`.

GRAPE(spline1d, "knots2bezier")()

SPLINE1D *spline1d

Generates the spline parameters from the knot positions and tangents.

GRAPE(spline1d, "list-alloc")(length)

SPLINE1D *spline1d
int length

Allocates memory for `length` knots (in other words `length-1` intervals).

GRAPE(spline1d, "list-free")()

SPLINE1D *spline1d

GRAPE(spline1d, "read")(file_name)

SPLINE1D *spline1d
char *file_name

Read a `.sp` file into an existing instance.

GRAPE(spline1d, "set-linear")(const1, const2, min_arg, max_arg, max_deg, mno_intervals)

SPLINE1D *spline1d
double const1, const2, min_arg, max_arg
int max_deg, mno_intervals

This method will first call the methods `"list-alloc"` — eventually calling `"list-free"` before — and use `max_deg` as the new maximal degree of the spline and `mno_intervals` as the new maximal number of intervals. The maximal degree of the spline must be 3, other values are not supported up to now. After allocating memory for the new spline, it will create a linear spline with the start knot (const1, min_arg) and the end knot (const2, max_arg).

GRAPE(spline1d, "set-value-for")(arg, val, eps, insert_new_knot_if_needed)

SPLINE1D *spline1d
double arg, val, eps
int insert_new_knot_if_needed

Does what it says.

GRAPE(spline1d, "write")(file_name)

SPLINE1D *spline1d
char *file_name

Write itself as a `.sp` file.

9.1.3.26 `Spline_Editor`

(subclass of 9.1.3.12 `Group`)
Stand alone editor for splines.

GRAPE(Spline_Editor, "new-instance")(name)

CLASS *Spline_Editor
char *name

Create a new Spline_Editor instance.

GRAPE(spline_editor, "set-spline")(spline)

SPLINE_EDITOR *spline_editor
SPLINE1D *spline

Set `spline` to be edited. If there is a spline already set it must be removed with the method `"unset-spline"`.

GRAPE(spline_editor, "unset-spline")()

SPLINE_EDITOR *spline_editor

Remove the link to a spline, the Spline_Editor will appear empty.

9.1.3.27 StaticText

(subclass of 9.1.3.29 `Textmessage`)

Nearly the same as Textmessage. The only difference is the better `"new-instance"` method which has a second parameter for the displayed text and initializes the border state to be `bfNoBorder`.

GRAPE(StaticText, "new-instance")(name, lines)

CLASS *StaticText
char *name, *lines

Creates a new StaticText instance named `name` with the label `lines`, without border. The text will be displayed left and vertically centered.

9.1.3.28 Step_Button

(subclass of 9.1.3.2 `Button`)

An autorepeating Button. If you press and hold a mouse button above a Step_Button it will repeatedly call its `action`.

To use a Step_Button simply create it by

```
button = (BUTTON *)GRAPE (Step_Button, "new-instance")
        (method, instance, "name");
```

instead of

```
button = (BUTTON *)GRAPE (Button, "new-instance")
        (method, instance, "name");
```

9.1.3.29 Textmessage

(subclass of 9.1.3.15 `Item`)

A simple static text (though the text can be changed while displayed). Allows to set different alignment modes (see below) and border styles.

If the displayed text — sometimes called label — contains `\n` characters it will have multiple lines.

```
typedef enum {
  afCenter,
  afTop,
  afBottom,
  afLeft,
  afRight
} ALIGN_FLAGS;
```

GRAPE(textmessage, "change-border")(border)

> **TEXTMESSAGE *textmessage**
> **BORDER_FLAG border**

Set the border style. A Textmessage will look like a pressed Button if `border` is set to `bfBorder`, it will look like simple text if bfNoBorder is used.

(char *) GRAPE(textmessage, "get-text")(buf, len)

> **TEXTMESSAGE *textmessage**
> **char *buf**
> **int len**

The caller supplied buffer `buf` (`char[len]`) is filled with the currently shown label.

GRAPE(Textmessage, "new-instance")(lines)

> **TEXTMESSAGE *textmessage**
> **char *lines**

Create a new instance. Name and label are set to be copies of `lines`.

If the name shall be distinct from the label use StaticText instead.

GRAPE(textmessage, "set-align")(alignment)

> **TEXTMESSAGE *textmessage**
> **ALIGN_FLAGS alignment**

Set a new text alignment inside the rectangular border. `alignment` is *(vertical alignment)* `|` *(horizontal alignment)*. Example: `(afCenter | afRight)`.

GRAPE(textmessage, "set-text")(text)

> **TEXTMESSAGE *textmessage**
> **char *text**

Use `text` as new label, i.e. as message.

9.1.3.30 `XList`

(subclass of 9.1.3.12 `Group`)

An Xlist is used to select one of several items. The content of the input can be completed using the `TAB` key. The XList will try to find a matching item in the list.

```
typedef enum sort_method {
  smUnsorted,
  smAlphabetic,
  smFileList,
  smMethodList
} SORT_METHOD;
```

GRAPE(xlist, "add-key-to-list")(newstr, key)

> **XLIST *xlist**
> **char *newstr**
> **int key**

The same as `"add-to-list"` except that the inserted string is associated with the number `key`. In the list method one can distinguish between similar entries by comparing the instance variable `xlist->chosen_key` instead on interpreting only `xlist->chosen`. Note that `xlist->chosen_key` is only valid if `xlist->chosen` is set.

GRAPE(xlist, "add-to-list")(newstr)

XLIST *xlist
char *newstr

Add the string `"newstr` to the list. If a sorting mode is set, using the method `"set-sorting-mode"`, the item will be added at the right position.

GRAPE(xlist, "focus-element")(element)

XLIST *xlist
char *element

Set the list element matching `element` to be chosen; it appears pressed in the list and the list is scrolled to show it.

GRAPE(xlist, "list-is-empty")()

XLIST *xlist

Return non `NULL`, if `xlist` has list entries, i.e. the list does not need to be rebuilt.

GRAPE(XList, "new-instance")(meth, inst, name)

CLASS *XList
char *meth
INSTANCE *inst
char *name

GRAPE(xlist, "remove-from-list")(str)

XLIST *xlist
char *str

Remove a list entry by string.

GRAPE(xlist, "remove-key-from-list")(key)

XLIST *xlist
int key

Remove a list entry by key.

GRAPE(xlist, "remove-list")()

XLIST *xlist

Empty the list, to be prepared for creating new entries.

GRAPE(xlist, "set-sorting-mode")(sorting_mode)

XLIST *xlist
SORT_METHOD sorting_mode

Set the kind of sorting, which is used only while adding items to the list.

GRAPE(xlist, "strip")(class)

XLIST *xlist
CLASS *class

Remove all entries of `xlist` that are of class `class`. This is the only way to remove unused `Inputs` or `Buttons`. `class` may be only `Input` or `Button` for this reason.

9.1.3.31 `XList_Button`

(subclass of 9.1.3.2 `Button`)

An XList_Button is a very special kind of Button, when it is created (using `"new-instance"`) an XList contained in a Layer is created as well. In its `action`, i.e. when it is clicked, the Button will show the Layer to allow using the XList; of course XList stands for any subclass of XList.

When the button is freed it will delete the Layer and the XList as well.

(XLIST_BUTTON *) GRAPE(XList_Button, "new-for-xlist")(meth, inst, name, get_data)

> **CLASS *XList_Button**
> **char *meth**
> **INSTANCE *inst**
> **char *name, *get_data**

This method creates a Button, which will pop-up an XList, if you press the button. The XList will then send the method defined in the parameter `"get-data"` to itself. This method has to fill the XList with data. If the user ends the selection of a list element using the XList's OK- or Cancel-button, the method `meth` will be send to the instance `inst`. This method must also hide the XList by sending the method `"hide"` to the XList's Layer.

(XLIST_BUTTON *) GRAPE(XList_Button, "new-for-filelist")(meth, inst, name, get_data, wildcard, open_mode)

> **CLASS *XList_Button**
> **char *meth**
> **INSTANCE *inst**
> **char *name, *get_data, *wildcard**
> **OPEN_MODE open_mode**

This method will create a button, which will pop-up a file-list, with the same principal as `"new-for-xlist"`. Additional to `"new-for-xlist"` you must define a wildcard, to select the files you want to see in the file-list. With the parameter `open_mode` you can define if the file should be opened for reading or for writing. Normally the `get_data` parameter should be `"get-files"`.

(XLIST_BUTTON *) GRAPE(XList_Button, "new-for-methlist")(meth, inst, name, get_data, end_of_meth, nr_of_additionals, additionals)

> **CLASS *XList_Button**
> **char *meth**
> **INSTANCE *inst**
> **char *name, *get_data, *end_of_meth**
> **int nr_of_additionals**
> **char **additionals**

This method creates a button, which will pop-up a method list with the same principal as `"new-for-xlist"`. If the value of `end_of_meth` is != NULL, only method, whose names have the correct ending will be added to the list, for example this `"-disp"` to get only display–methods. With the additional parameters a number of methods can be added to the list, which don't need to have the correct ending, for example `"display"` to add the default-display method to the list of display methods. Normally the `get_data` parameter should be `"get-methods"`.

GRAPE(butt, "set-label")(label)

> **XLIST_BUTTON *butt**
> **char *label**

Like `"set-label"` on the superclass `Interactive`, but the `label` is used for the assiciated `XList` as well.

GRAPE(butt, "share-path")(other)

XLIST_BUTTON *butt, *other

If `butt` and `other` have been created by `"new-for-filelist"`, the associated `FileList`s share their working directory hereafter.

9.1.4 Utility Functions

There are some utility functions to open often used layers on the standard manager.

`int` **g_msgbox** `(char *title, char *text, int buttons)`
: Creates and opens a messagebox with a given title and the text `text`, containing the buttons `buttons`, which is a number `OR`ed together from the flags `mbfYes`, `mbfNo`, `mbfCancel`, `mbfOk` and `mbfAbort`. If buttons is `mbfNone` it closes automatically after a few seconds. If `title` is `NULL` a default value is used. For multiline messages insert `\n` where you want to break the line. The function returns the flag of the pressed button or `0` if an error occurred.

  The handle is blocked while the messagebox is active.

`void` **g_infobox** `(char *title, char *text)`
: Creates and opens a non blocking infobox with the given `title` and `text`. If `title` is `NULL` a default title is used. For multiline messages insert `\n` where you want to break the line.

  The box will remain open as long as `dismiss` is not pressed.

`ALERT(condition, message, errorhandler)`
: `ALERT` is the interactive replacement of `ASSURE`. If `condition` equals `0` a `g_msgbox` with message `message` is opened. After having confirmed, `errorhandler` is executed.

  As with `ASSURE`, `errorhandler` may consist from more than one statement, separated by `";"`.

Some other variables and functions.

`void` **g_hide_layer** `(ITEM *item)`
: Hides the layer the item belongs to.

`double` **g_eval_ident_fn** `(double x)`

`double` **g_eval_angle_fn** `(double x)`

`double` **g_eval_exp_fn** `(double x)`

`double` **g_eval_log_fn** `(double x)`

`double` **g_eval_exp10_fn** `(double x)`

`double` **g_eval_log10_fn** `(double x)`
: Predefined functions to be used as evaluations functions for `Bar1d` instances.

`int g_batch_mode`
: `TRUE` if **GRAPE** runs in batch mode. Set this before calling **GRAPE** if the program shall not open windows, e.g. if running on a batch server. On the other hand do not call functions or methods that require or create a manager, especially do not call `GRAPE(Manager, "get-stdmgr")()` and do not create any `Item`s which will try get the standard manager.

  Do not change the flag after having made the first `GRAPE` call.

Changes from 5.0:

```
#G_HEAD #define msgbox g_msgbox
#G_HEAD #define infobox g_infobox
#G_HEAD #define G_MSGBOX_BUTTON_YES    mbfYes
#G_HEAD #define G_MSGBOX_BUTTON_NO     mbfNo
#G_HEAD #define G_MSGBOX_BUTTON_CANCEL mbfCancel
#G_HEAD #define G_MSGBOX_BUTTON_OK     mbfOk
#G_HEAD #define G_MSGBOX_BUTTON_ABORT  mbfAbort
#G_HEAD #define G_MSGBOX_RECURSIVE     mbfRecursive
#G_HEAD #define dfINT dfInt
#G_HEAD #define dfFLOAT dfFloat
#G_HEAD #define dfDOUBLE dfDouble
#G_HEAD #define dfSTRING dfString
#G_HEAD #define dfREGULAR_EXPRESSION dfRegexp
```

9.2 Graphic Devices

It is intended to develop the graphic environment **GRAPE**independent of any hardware requirements. To accomplish this aim the class GraphicDevice is introduced. Its instances provide primitive functions for graphic output.

GRAPE(dev, "set-stddev")()

(GRAPHICDEVICE *) GRAPE(GraphicDevice, "get-stddev")()

(GRAPHICDEVICE *) GRAPE(special-dev-class, "new-instance") (name)

> **GRAPHICDEVICE *dev**
> **CLASS *GraphicDevice, *special-dev-class**
> **char *name**

Graphic output is always sent to the actual device (or `stddev`). On class GraphicDevice a method `"get-stddev"` is defined which returns a pointer on the actual standard device (`stddev`). On the other hand a subclass' instance of GraphicDevice can be made to the standard device by sending the method `"set-stddev"`. Since no device is explicitly set the graphic output is sent to a default device depending on the user's machine and terminal.

The method `"new-instance"` creates a new instance of the available subclasses. The only parameter defines its name. The function pointers are described later. This method does not change the actual graphic device. The actual graphic device is set by `"set-stddev"`, e.g.:

```
GRAPHICGT *dev;

dev = GRAPE(GraphicGt, "new-instance")("graphic-out");
GRAPE(dev, "set-stddev")();
```

9.2.1 Device

The class GraphicDevice is a subclass of Device. The superclass Device supplies mainly the instance variable `RW_flags` defining the device's capabilities for reading, writing or both.

The class Device is defined as follows:

```
#CLASS Device : Root {
  unsigned RW_flags;            /* read/write possible? */
};
```

The variable `RW_flags` can hold the following values but the value should only changed by methods on class Device and not by the programmer.

```
G_READONLY
G_WRITEONLY
G_READWRITE
```

9.2.2 GraphicDevice

The class GraphicDevice is only a superclass for the different classes representing the different hardware platforms. Recently implemented subclasses are

- GraphicOgl: OpenGL
- GraphicX11: X Window System, Version 11
- GraphicPs: Postscript, to create an (encapsulated) postscript file
- GraphicRay: Raytracing, to create a file for public domain rayshade
- GraphicSoft: Raytracing/Animation (to create files for SOFTIMAGE). This driver is not contained in the standard **GRAPE**–library, SOFTIMAGE users will get the information when asking.
- GraphicGt: IRIS GL

The class GraphicDevice defines all pointers to the primitive functions containing the graphic output routines. The instance variable `info` points to a machine dependent structure containing the device's static variables, it is for internal use only.

As the graphics interface is one of the main parts of **GRAPE**, and should support any new hardware features, it is constantly worked on, so many of its parts are still in experimental state. The main functions like `move`, `draw`, `begin_patch`, `patch_vertex`, `patch_normal`, `end_patch`, `transform`, `attribute`, the light sources, are, however, stable and non–changing.

The structure of the class `GraphicDevice` is defined as follows:

```
#CLASS GraphicDevice : Device {
  int *info;              /* pointer to device specific informations */
  void (*update)(void);   /* this has to be the first pointer */
  void (*clear)(void);
  void (*message)(long, char *);
  int  (*attribute)(int, int, void *);
  void (*transform)(int, int, MATRIX44);
  void (*move)(VEC3);
  void (*draw)(VEC3);
  void (*begin_patch)(void);
  void (*patch_vertex)(VEC3);
  void (*patch_vertex_index)(int);
  void (*patch_normal)(VEC3);
  void (*patch_color)(VEC3);
  void (*end_patch)(void);
  void (*move_direct)(VEC3);             /* internal use only */
  void (*draw_direct)(VEC3);             /* internal use only */
  void (*begin_patch_direct)(void);      /* internal use only */
  void (*patch_vertex_direct)(VEC3);     /* internal use only */
  void (*patch_normal_direct)(VEC3);     /* internal use only */
  void (*patch_color_direct)(VEC3);      /* internal use only */
  void (*end_patch_direct)(void);        /* internal use only */
  void (*color_alpha)(VEC3);
  void (*texture_init)(unsigned char *,unsigned int, int, int);
```

```
  void (*texture_coord)(double, double);
  int (*begin_list)(void);      /* These Display-List functions are used for */
  void (*end_list)(void);       /* small temporary lists like a list for a   */
  int (*call_list)(int);        /* isocaeder. They should be deleted at the  */
  void (*delete_list)(int);     /* end of the display-method.                */
  int  (*generate_lightsource)(int);
  void (*lightsource)(int, int, LIGHTSOURCE_DEV *);
  void (*delete_lightsource)(int);
  void (*text)(VEC3, char *);
  void (*set_full_screen)(int);
  void (*video_record_function)(void);

  int grid_patch;        /* G_PATCH patch is available; G_GRID always grid */
  int interactive;       /* OFF handle not available: else ON */
  int global_grid_patch; /* supervises grid_patch */
 } GRAPHICDEVICE;
```

A program has access to the primitive graphic routines for the actual device if it demands a pointer to the standard device by the command line

```
dev = (GRAPHICDEVICE *) GRAPE(GraphicDevice, "get-stddev")();
```

The pointer has to be updated if the standard device might have changed. The variables must not be changed by hand and even the function pointers can change when switching methods are called.

Now a brief description for the primitive graphic routines:

9.2.2.1 Primitive Graphic Routines

These are the basic functions for creating lines and patches. Functions like `attribute` or `transform` would fit in this list, but they have a global character as well.

`void` **move** `(VEC3 point)`
: The line drawing pen position is set to `point`.

`void` **draw** `(VEC3 point)`
: Draw a line from the current pen position to the position `point` and update the pen position. The line regards type and color settings.

`void` **begin_patch** `(void)`
: This starts the definition of a patch. The patch's border is a polygon. Each line is defined by vertices and each vertex may have a normal.

`void` **patch_vertex** `(VEC3 point)`
: This defines the next vertex of the patch. This vertex point regards the current patch attributes (e.g. normal, color, ...).

`void` **patch_normal** `(VEC3 normal)`
: This defines the normal for all patch vertices following before the next call to this function. If the attribute `G_LIGHT_MODEL` is set to `ON` this normal is taken into account to calculate the lighting.

`void` **patch_color** `(VEC3 color)`
: This defines the color for the following patch vertices. The setting is changed by another call to this function. It should be used only in between `dev->begin_patch` and `dev->end_patch`.

`void` **end_patch** `(void)`
: This closes the definition of the actual patch.

void text (VEC3 point, char *string)
Displays the 2–D text `string` at the position `point` in three dimensional coordinates regarding the current text attributes for color, font, etc. The `point` will be transformed into screen coordinates by the actual matrices. It is machine dependent if the transformation has to be performed by the device driver.

9.2.2.2 Global Settings and Matrix Handling

void update (void)
All data will be put on the screen including buffered data. Therefore the screen will display the actual picture.

void clear (void)
The whole screen will display the actual background color (see `dev->attribute()` below how to set the background color).

int attribute (int mode, int type, void *data)
Query, get or set an attribute. In query mode the data item will be ignored. The data item might be casted to the appropriate type. For a detailed description see 9.2.2.6.

void transform (int mode, int type, MATRIX44 matrix)
`mode` is one of `G_MODE_GET`, `G_MODE_SET`, `G_MODE_PRE`, `G_MODE_POST`, `type` is one of `G_MATRIX_PROJECT`, `G_MATRIX_VIEW`, `G_MATRIX_MODEL`, `G_MATRIX_PVM`, `G_MATRIX_IVM` (the special value `G_MATRIX_NONE` is not used here).

Handling of transformation matrices. The transformation from model to screen coordinates is performed in several steps. The `MODEL` matrix changes the model coordinates to world coordinates, the `VIEW` matrix continues the transformation to normalized coordinates and the `PROJECT` matrix to standard coordinates. In the standard system the visible region is the cube [–1, 1] x [–1, 1] x [–1, 1]. Finally a machine dependent transformation into screen coordinates will be performed.

The matrices will be multiplied to the left of the vectors, i.e. the precedence is (`PROJECT * VIEW * MODEL`) * vector. The `MODEL` matrix's duty is the generation of a scene containing primitive objects, the `VIEW` matrix concludes the viewer's position and orientation and the `PROJECT` matrix creates a suitable perspective.

The mode `G_MODE_GET` is useful to read the desired matrix specified by the `type`. The mode `G_MODE_SET` affects the data of the specified `matrix`. The modes `G_MODE_PRE` and `G_MODE_POST` are intended to multiply a new matrix from left (post) or right (pre) to an existing matrix. The matrices `PVM` and `IVM` can only be read. Here `PVM` is equal to (`PROJECT * VIEW * MODEL`) and `IVM` is equal to the inverse of (`VIEW * MODEL`) but only the upper 3 x 3 entries are taken into account. The transponent of `IVM` transforms the normal vectors.

9.2.2.3 Display Lists

If display lists are supported by the hardware (for instance on advanced OpenGL machines), or when a remote display is used and the lists are stored locally, they can speed up the drawing dramatically. Even on the other devices, where the display lists are emulated by software, they can reduce repeated calls of slow display methods.

There is no display list management for whole objects and data hierarchy trees at the moment (due to problems caused by local scene matrices), and it would be no good idea to implement display lists shared between arbitrary `GraphicDevice` subclasses. The one thing possible are local display lists that are allocated at the startup of a display method, then used, and freed at the end. As mentioned before they cannot be used across calls of display methods, because these could happen on different devices. One application is to create a display lists containing the shape of a sphere or anything else and to use it many times at once, moved by a pre transformation of the MODEL matrix, to display a set of points.

`int` **begin_list** `(void)`
Begin a list. All function calls up to the next `end_list` are recorded. The list-ID is returned, or 0 in case of an error. Calls of `begin_list` cannot be nested, but lists can contain invocations of other lists.

`void` **end_list** `(void)`
Finish recording a list.

`int` **call_list** `(int list_id)`
Draw the contents of the list with the gived ID.

`void` **delete_list** `(int list_id)`
Delete the given list.

9.2.2.4 Texturing

The texture functions are experimental and only implemented in the OpenGL driver for now, in a somehow hardware dependent way. So do not use them if you are interested in images with a higher than screen resolution.

`void` **texture_init** `(unsigned char *, unsigned int, int, int)`
This is used to define a texture by `dev->texture_init (&Image[0][0][0], G_RGB, imageWidth, imageHeight);` here `Image` is a matrix of size `imagewidth`*`imageheight`*3: `unsigned char Image[imagewidth][imageheight][3];` the integers `imagewidth`, `imageheight` are the width and the height of the image. Notice that `imagewidth`, `imageheight` are numbers of the form 2^m and the smallest number is 64. For better use take `imagewidth` = `imageheight`.

`void` **texture_coord** `(double, double)`
This function has to be called during patch generation, i.e. between subsequent calls of `patch_vertex()`. After defining a texture with `void (*texture_init)` you can define a point in texture space $\in \mathbb{R}^2$ with `dev->texture_coord(xp,yp);` here `xp`, `yp` are double values between 0.0 and 1.0. Then you define a point in your image space $\in \mathbb{R}^3$ with `dev->patch_vertex(pkt);` here `pkt` is of type `VEC3`.

9.2.2.5 Lightsource Handling

`int` **generate_lightsource** `(int type)`

```
/* types */
G_LIGHT_AMBIENT
G_LIGHT_DIRECT
G_LIGHT_POINT
G_LIGHT_SPOT
```

This defines a new light source. The available types are device dependent. If no light source of the desired type is available the return code is zero. The different types are

AMBIENT	ambient light, without any direction
DIRECT	directed light with source located at infinity
POINT	light is coming from one point and sent in all direction
SPOT	light from one point is sent in a certain direction and angle

Not all types are supplied by all machines. Since no more attributes are available for the light sources they are switched off.

```
    /* modes */
    G_MODE_GET
    G_MODE_SET

typedef struct lightsource_dev {
    int type;           /* G_LIGHT_AMBIENT | DIRECT | POINT | SPOT */
    int on_off;         /* lightsource switched ON or OFF */
    double color[3];    /* rgb light color (incl. intensity) */
    double position[3]; /* POINT | SPOT light source position */
    double direction[3]; /* DIRECT | SPOT light direction */
    double open_angle;  /* SPOT open angle */
} LIGHTSOURCE_DEV;
```

`void` **lightsource** `(int mode, int light, LIGHTSOURCE_DEV *data)`
Read or modify any light source. The light's number `light` has to be returned by a previous `dev->generate_lightsource()`. A light source is maintained by its `data` but its type can not be changed by `G_MODE_SET`. It has been defined by `dev->generate_lightsource()`.

`void` **delete_lightsource** `(int light)`
This removes the given light source `light` previously defined by `dev ->generate_lightsource()`.

9.2.2.6 The `attribute` Function

`int` **attribute** `(int mode, int type, void *data)`
The mode–parameter may have the following values, but not every mode and type will work together. For the correct specification see the appropriate table. The modes PRE and POST are used in the transformation function and only mentioned here for completeness.

```
/* modes */
G_MODE_QUERY
G_MODE_GET
G_MODE_SET
G_MODE_PRE
G_MODE_POST
G_MODE_SWAP
```

This paragraph describes the possible modes and their functionality:

- **G_MODE_QUERY** This mode will return exactly one integer value, whereas zero equals 'no' or 'off', one equals 'yes' or 'on' and minus one represents an error condition. The third item data is ignored at all.
- **G_MODE_GET** In this mode the demanded data will be given back by the third item, which is casted to the correct data type. On successful completion the return value is zero, otherwise minus one.
- **G_MODE_SET** In this mode the attribute will be set to the given data. Same error handling as in G_MODE_GET.
- **G_MODE_SWAP** A special mode, which is only used for the attribute "double-buffer" and described there.

Now the list of attributes, which can be demanded or set by the `attribute()` function. The attributes are not only depending upon the graphic–hardware, but mostly on the device–driver.

```
G_RASTER_ABILITY
G_COLOR_ABILITY
```

```
G_ZBUFF_ABILITY
G_RESOLUTION
G_ASPECTRATIO
G_BACKGROUND_COLOR
G_LINE_COLOR
G_LINE_TYPE
G_ANTIALIAS
G_DEPTHCUE
G_PATCH_SUPROP
G_TEXT_COLOR
G_TEXT_TYPE
G_DOUBLEBUFFER
G_LIGHT_MODEL
G_LIGHT_NUMBER
G_HAS_DOUBLEBUFFER
G_ORIGIN
G_TEXTURE
```

The following table shows the possible combinations of modes, types and the appropriate data-type needed. (— means data ignored, return value of one means yes)

```
attribute          modes        *data                comment
---------          -----        -----                -------
G_RASTER_ABILITY   QUERY          -                  hardw. for raster graphic
G_COLOR_ABILITY    QUERY          -                  hardware for colors
G_ZBUFF_ABILITY    QUERY          -                  hardware z-buffering
G_RESOLUTION       GET | SET    RESOLUTION_TYPE xy   size of window in pixel
G_ASPECTRATIO      GET          double aspect        win_size_x / win_size_y
                                                     (only 1.0 for now)
G_BACKGROUND_COLOR GET | SET    VEC3 rgb
G_LINE_COLOR       GET | SET    VEC3 rgb
G_LINE_TYPE        GET | SET    int itype            (only X11: default = 0,
                                                      dashed < 0 < thickness)
G_ANTIALIAS        GET | SET    int yes_no
G_DEPTHCUE         GET | SET    DEPTHCUE_TYPE depthcue
G_PATCH_SUPROP     GET | SET    SUPROP_DEV suprop
G_TEXT_COLOR       GET | SET    VEC3 rgb
G_TEXT_TYPE         -             -                  (not implemented)
G_DOUBLEBUFFER     GET|SET|SWAP int yes_no
G_LIGHT_MODEL      SET          int on_off           inhibit light calculation
G_LIGHT_NUMBER     QUERY          -                  number of light sources
G_HAS_DOUBLEBUFFER  -             -                  (not implemented)
G_ORIGIN           GET | SET    RESOLUTION_TYPE xy   position of window
```

In the table above the following typedefs are used:

```
int on_off, yes_no;          /*  NO, OFF or YES, ON */

typedef struct suprop_dev {
 double emission[3];  /* rgb color emission (incl. intensity) */
 double ambient[3];   /* ambient rgb reflectance             */
 double diffuse[3];   /* diffuse rgb reflectance             */
 double specular[3];  /* specular rgb reflectance            */
 double specular_exp; /* specular reflectance exponent       */
 double transparency; /* transparency factor (0..1)          */
} SUPROP_DEV;

typedef struct depthcue_type {
 int on_off; /* OFF, ON */
```

```
  double zmin; /* dist. where depth cueing begins (full intensity) */
  double zmax; /* dist. where depth cueing ends (reduced intensity)*/
  double dark; /* max. reduction of intensity (0.0 <= dark <= 1.0) */
 } DEPTHCUE_TYPE;
```

9.2.2.7 Methods on GraphicDevice

The most important methods on `GraphicDevice` are described at the beginning of section 9.2.

(GRAPHICDEVICE *) GRAPE(dev1, "copy")(dev2, name)

GRAPHICDEVICE *dev1, *dev2
char *name

This method copies the usual data (matrices etc.) of the graphic device `dev1` to `dev2` if this is an instance. If `dev2` is a subclass (e.g. `GraphicGt`) of `GraphicDevice` a new instance of this class will be created with the pointer `dev2`. The named instance's return value is `dev2`.

(int *) GRAPE(dev, "grid-patch")()

(int *) GRAPE(dev, "global-grid-patch")()

(int *) GRAPE(dev, "set-grid-patch")(mode)

(int *) GRAPE(dev, "set-glbl-gr-ptch")(mode)

GRAPHICDEVICE *dev
int mode

Any graphic device has two different modes. If patches are display able they will be displayed shaded with normals at the vertices or displayed flat without normal information. Otherwise only wire frames are displayed.

The current mode is hold in the device instance variable `grid_patch`. The values are `G_GRID` for wire frames and `G_PATCH` for patch models. The variable `grid_patch` may be read. To change the mode use the method `"grid-patch"` which return value is the new mode. An unconditional setting of the mode is done by `"set-grid-patch"` and not by assignment at all.

If the global value `global_grid_patch` is set to `G_GRID`, `"set-grid-patch"` will not switch to another mode then `G_GRID`. This is because of the implementation of the `GraphicDevice` subclasses which have no hardware 3D support: they will work very much faster if no patches are drawn.

GRAPE(dev, "get-pick-pos")(pickx, picky, pickkey)

GRAPHICDEVICE *dev
double *pickx, *picky
int *pickkey

The coordinates of the current mouse position in the actual graphic device will be stored in `*pickx` and `*picky` in the range of -1.0 and 1.0. `*pickkey` holds the actual mouse or keyboard event. Unfortunately the values are coded in decimal. The codes are:

1001	left mouse button pressed
1002	right mouse button pressed
1003	middle mouse button pressed
2001	left mouse button released
2002	right mouse button released
2003	middle mouse button released
else	ASCII value of the key pressed

If the mouse does not point into the graphic window `pickkey` is set to zero, but `pickx`, `picky` have any values.

(LAYER *)GRAPE(dev, "create-config")()

GRAPHICDEVICE *dev

Create a `Layer` for editing device specific values.

If this method is supplied by a subclass of `GraphicDevice`, one can fine tune its settings.

GRAPE(dev, "show-lights")()

GRAPHICDEVICE *dev

Display balls and arrows for the lightsources. This method is used by the light menu `show lights` option.

GRAPE(dev, "supported")(display)

GRAPHICDEVICE *dev
char *display

Returns non `NULL` if the display described by `display` supports this subclass of `GraphicDevice`. This does not work well with the class `GraphicGt`, which is the reason why users of this class have to supply environment strings for this purpose.

9.2.3 An Example Low–Level Display Method

```
TRIANG2D *triang2d_example_disp (void)
/* this method is not optimized in any way. */
{
  TRIANG2D *self;
  GRAPHICDEVICE *grdev;
  MANAGER *mgr;
  static int own_colors = 0;
  static CHECKBOX *checkbox = NULL;

  self = (TRIANG2D *) START_METHOD (G_INSTANCE);
  ASSURE (self, "", END_METHOD (NULL));

  mgr = (MANAGER *) GRAPE (Manager, "get-stdmgr")();
  ASSURE (mgr, "triang2d_example_disp: can't get manager",
          END_METHOD (NULL));


  /* we have a switch: "own_colors".
   * to let the user choose a value we use a Checkbox.
   */
  if (!checkbox) {
    /* no checkbox yet: create it */
    checkbox = (CHECKBOX *) GRAPE (Checkbox, "new-instance")
      ("example: colors", &own_colors);
    GRAPE (checkbox, "set-pref-size")(6.0, 1.0);
  }

  /* insert the checkbox into menu "opt 0" if not already there.
   * it will be removed when this method is not used any more
   */
  if (GRAPE (mgr, "new-handle")(triang2d_example_disp, 1)) {
    GRAPE (mgr, "add-inter")(checkbox);
  }
```

```
/* get standard graphic device.
 * create error message when fail.
 */
grdev = (GRAPHICDEVICE *) GRAPE (GraphicDevice, "get-stddev")();
ASSURE (grdev, "triang2d_example_disp: can't get GraphicDevice",
        END_METHOD (NULL));



if (grdev->grid_patch == G_GRID) {
  /* Device is switched to grid mode. We MUST NOT draw patches
   * here (it may work on your device, but not on others, e.g.
   * X11, PostScript ...; to have patch-only and mixed modes
   * alternatively you have to use patch mode and extra
   * selfmade switches).
   */
  if (!own_colors) {
    /* normally use the preset line color */
    int i, j;
    VEC3 point[3];
    for (i = 0; i < self->number_of_elements; i++) {
      for (j = 0; j < 3; j++) {
        point[j][0] = self->x[self->vertex[i][j]];
        point[j][1] = self->y[self->vertex[i][j]];
        point[j][2] = self->z[self->vertex[i][j]];
      }
      grdev->move (point[0]);
      for (j = 0; j < 3; j++)
        grdev->draw (point[2 - j]);
    }
  }

  else {
    /* special: we use our own line color: Red */
    int i, j;
    VEC3 point[3], linecolor;
    static VEC3 Red = { 1.0, 0.0, 0.0};

    /* preserve the previously set line color */
    grdev->attribute (G_MODE_GET, G_LINE_COLOR, linecolor);
    grdev->attribute (G_MODE_SET, G_LINE_COLOR, Red);
    for (i = 0; i < self->number_of_elements; i++) {
      for (j = 0; j < 3; j++) {
        point[j][0] = self->x[self->vertex[i][j]];
        point[j][1] = self->y[self->vertex[i][j]];
        point[j][2] = self->z[self->vertex[i][j]];
      }
      grdev->move (point[0]);
      for (j = 0; j < 3; j++)
        grdev->draw (point[2 - j]);
    }
    /* set the previously set line color */
    grdev->attribute (G_MODE_SET, G_LINE_COLOR, linecolor);
  }

}
```

```
else /* (grdev->grid_patch == G_PATCH) */ {
  /* Device is switched to patch mode. We can draw patches or
   * grids and mix it.
   */

  /* In this example we always use patches here. Normally we
   * want them to be colored automatically with respect to
   * the lightsources.
   * here another type of display is shown as well: selfmade colors
   */
  if (!own_colors) {
    int i, j;
    VEC3 point, normal;

    /* we use the lightsources */

    for (i = 0; i < self->number_of_elements; i++) {
      grdev->begin_patch();
      for (j = 0; j < 3; j++) {
        point[0] = self->x[self->vertex[i][j]];
        point[1] = self->y[self->vertex[i][j]];
        point[2] = self->z[self->vertex[i][j]];

        /* calculate the patch normal in this vertex from your data
           normal = my_normal (self->vertex[i][j]);

           the default display method takes the average normal of the
           triangles using this vertex

           we take a very stupid formula here */
        normal[0] = g_sin (point[2]);
        normal[1] = g_cos (point[2]);
        normal[2] = point[2];
        g_vec3_normalize (normal);

        grdev->patch_normal (normal);
        grdev->patch_vertex (point);
      }
      grdev->end_patch();
    }
  }


  else {
    int i, j;
    int light_model, off = 0;
    VEC3 point, color;

    /* we use selfmade colors. switch off the light-model, but
     * save previous state to restore afterwards
     */
    grdev->attribute (G_MODE_GET, G_LIGHT_MODEL, &light_model);
    grdev->attribute (G_MODE_SET, G_LIGHT_MODEL, &off);

    for (i = 0; i < self->number_of_elements; i++) {
      grdev->begin_patch();
```

```
        for (j = 0; j < 3; j++) {
          point[0] = self->x[self->vertex[i][j]];
          point[1] = self->y[self->vertex[i][j]];
          point[2] = self->z[self->vertex[i][j]];

          /* calculate the color for this vertex from your data
             color = my_color (self->vertex[i][j]);

             we take a very stupid formula here */
          color[0] = g_sin (10. * point[2]);
          color[1] = g_cos (10. * point[1]);
          color[2] = g_sin (10. * point[0]);

          grdev->patch_color (color);
          grdev->patch_vertex (point);
        }
        grdev->end_patch();
      }
      grdev->attribute (G_MODE_SET, G_LIGHT_MODEL, &light_model);
    }

  }

  END_METHOD (self);
}
```

9.3 Resource file .graperc

At start GRAPE reads files `~/.graperc` and `./.graperc`. Their contents are accessible through the standard Instance of the class `ResourceDev`, which implements a multi purpose interface for classes to read structured user defined values of many types. As you will see you can use `ResourceDev` instances for applications other than `.graperc`-files, for instance for formula script files.

Currently this option is used by only two classes, giving an insight into the strength of this concept: you can save a lot of time when you specified default settings and keyboard macros, which is done by simply describing the user interaction. Since the `ResourceDev` interface is object oriented every class inherits the tags of its superclasses (not yet as there are no superclasses using it).

The `.graperc` is read at GRAPE startup and will never be updated (since the classes using `.graperc` entries not will reread them).

9.3.1 General usage of ResourceDev files

A resource file is a plain ASCII file in the following syntax:

`#` — Comment. Every line beginning with a `#` is commented out.

`\` — Line continuation as in C.

`ident;` — Assign integer/real value `1` to identifier `ident`. Everywhere `ident = n;` is allowed you can use `ident;` for `n` = `1`.

`ident = n0, n1, n2, ...;` — Assign integer/real values `n0, n1, n2, ...` to identifier `ident`.

`ident = "xyz", "XYZ", ...;` — Assign string values `ident = "xyz", "XYZ", ...` to identifier `ident`. As in C `"ab" "cd"` (= `"abcd"`), `"\n\t\012\x12"` are valid methods to write strings.

`ident = { ... }` — Assign a group to identifier `ident`. Its contents are of the prescribed syntax so groups can be nested recursively.

`INCLUDE = "xyz";` — Include the contents of a group defined anywhere else.

Its interpretation is done by the calling class.

The identifiers of nested groups are written as a path.

9.3.2 Methods on class `ResourceDev`

(RESOURCEDEV *) GRAPE(ResourceDev, "new-instance")(name)

CLASS *ResourceDev
char *name

Create a new `ResourceDev` instance.

GRAPE(obj, "free")()

RESOURCEDEV *obj

Free an old, used up instance.

GRAPE(obj, "init")()

RESOURCEDEV *obj

Read the (set or added) resource files.

(RESOURCEDEV *) GRAPE(ResourceDev, "get-stddev")(name)

CLASS *ResourceDev

Return the standard `ResourceDev` instance. If none is set one is created and initialized by calling `"set-filename"("~/.graperc")` and `"append-filename"("./.graperc")`, then set as standard device and returned.

GRAPE(obj, "set-stddev")()

RESOURCEDEV *obj

Set `obj` as the standard device.

GRAPE(obj, "parse")(path, inst, meth, private)

RESOURCEDEV *obj
char *path
INSTANCE *inst
char *meth
void *private

Parse every line in the resource files whose group path matches path.

Special characters in the path:

`"/..."` — start at file level.

`"..."` — start with current path.

`"=..."` — read to end of current group only. Normally every matching line in the file is used, what is bad if you want to discriminate multiple local groups like in:

```
Manager = { hotkey = {} hotkey = {}}
```

.

For every resource `GRAPE(inst, meth)(descr, private)` is called, the resource value `descr` is passed in the following structure:

```
typedef struct g_resource_descr {
  char *ident;
  void *value;
  DECLARATION_FORMAT type;
  int index;
  int count;
  int sep, last_sep;
} G_RESOURCE_DESCR;
```

GRAPE(obj, "set-filename")(name)

RESOURCEDEV *obj
char *name

Delete all existing filenames, set `name` as name of resource file.

GRAPE(obj, "append-filename")(name)

RESOURCEDEV *obj
char *name

Add another filename.

GRAPE(obj, "print-error")(msg, res)

RESOURCEDEV *obj
char *msg, **res

Print a detailed error message including the filename and position in file, with description `msg`. Always use `res` = `NULL` in application code.

9.3.3 Utility Functions on class ResourceDev

`void` **g_remove_resourcelist**`(G_LIST *rlist)`
Remove all entries of the resource list `rlist` and the list itself.

`void` **g_resourcelist_print**`(G_LIST *rlist)`
Print all entries of rlist, used for the `"help"` tags.

`int` **g_resourcelist_find**`(G_LIST *rlist, G_RESOURCE_DESCR *descr, int *number)`
If `descr` matches on of the entries in `rlist` the value is copied to the location passed with `g_rclist_add`.

`G_LIST *`**g_rclist_add**`(G_LIST *rlist, char *ident, void *valueptr, DECLARATION_FORMAT type)`
Add a single value (which `valueptr` points to) of type `type` to the resource list, having the identifier `ident`. If `NULL` is passed as `rlist`, a `G_List` instance is created, which is returned.

`void` **g_resourcedev_not_found**`(INSTANCE *inst, G_RESOURCE_DESCR *descr)`
Print "not found" error message for current resourcedev.

`void` **g_resourcedev_parse**`(INSTANCE *inst, char *path)`
Easy interface for `"parse"`, requiring a method `"resource"`, and an optional method `"get-rclist"` on the class of `inst`.

9.3.4 Code Example : Resource Device Implementation in the Postscript Driver

```
GRAPHICPS *graphicps_resource (G_RESOURCE_DESCR *descr, G_LIST *rclist)
{
  GRAPHICPS *self;
  G_INFO_PS *info;
  int error = FALSE, used = FALSE, pass = FALSE;

  self = (GRAPHICPS *) START_METHOD (G_INSTANCE);
  ASSURE (self, "", END_METHOD (NULL));

  info = (G_INFO_PS *)self->info;

  /* special command: help prints all possible values */
  if (!strcmp (descr->ident, "help")) {
    used = TRUE;
    pass = TRUE;
    printf ("Help for section \"/GraphicPs\":\n"
            "The simple values are:\n");
    g_resourcelist_print (rclist);
    printf ("Additionally available:\n"
            "\trgb_to_gray of type VEC3 value %f %f %f\n"
            "\tfilename of type string value %s\n"
            "\thelp\n",
            info->rgb_to_gray_scale[0],
            info->rgb_to_gray_scale[1],
            info->rgb_to_gray_scale[2],
            info->file_name);
  }

  /* first try to resolve simple single values */
  if (!error && !used)
    used = g_resourcelist_find (rclist, descr, &error);

  /* none of them: hand-code the rest */
  if (!error && !used)
    switch (descr->type) {
    case dfGroup:
      /* no groups defined for now in GraphicPs */
      break;
    case dfDouble:
      if (!strcmp (descr->ident, "rgb_to_gray")) {
        static int rgb_to_gray_next = 0;
        /* vector handling not very good for now, wait for further revisions
         * of rcdev; don't use vectors before that
         */
        if (rgb_to_gray_next == descr->index) {
          info->rgb_to_gray_scale[descr->index] = *(double *)descr->value;
          rgb_to_gray_next = (descr->index + 1) % 3;
          used = TRUE;
        } else {
          error = TRUE;
          fprintf (stderr, "rgb_to_gray: index error\n");
        }
      }
      break;
    case dfString:
```

```
        if (!strcmp (descr->ident, "filename"))
          if (!descr->index && strlen (descr->value) < (size_t) MAX_STRLEN) {
            GRAPE (self, "set-filename")(descr->value);
            used = TRUE;
          }
          else
            error = TRUE;
        break;
      default:
        break;
      }

    if ((pass || (!error && !used)) &&
        GRAPE (self, "find-method")("^resource"))
      GRAPE (self, "^resource")(descr, NULL);

    if (error || !used) {
      g_resourcedev_not_found ((INSTANCE *)self, descr);
      END_METHOD (NULL);
    }
    END_METHOD (self);
  }

  G_LIST *graphicps_get_rclist (void)
  {
    GRAPHICPS *self;
    G_LIST *rclist = NULL;
    G_INFO_PS *info;

    self = (GRAPHICPS *) START_METHOD (G_INSTANCE);
    ASSURE (self, "", END_METHOD (NULL));

    if (GRAPE (self, "find-method")("^get-rclist"))
      rclist = (G_LIST *)GRAPE (self, "^get-rclist")();

    /* list must be created now because the variable addresses are not known at
     * compile/load time.
     */
    info = (G_INFO_PS *)self->info;

    rclist = g_rclist_add (rclist, "use_colors", &info->use_colors, dfInt);
    g_rclist_add (rclist, "patch_bounds", &info->use_patch_bounds, dfInt);
    g_rclist_add (rclist, "optimal_rotate", &info->optimal_rotate, dfInt);
    g_rclist_add (rclist, "optimal_stretch", &info->optimal_stretch, dfInt);
    g_rclist_add (rclist, "linewidth", &info->linewidth, dfDouble);
    [...]

    END_METHOD (rclist);
  }

  static void read_graperc (GRAPHICPS *self)
  {
    g_resourcedev_parse ((INSTANCE *)self, "/GraphicPs");
  }
```

The class `Manager` uses sub groups for the hotkeys:

```
  static int hotkey_has_key = FALSE;
```

```
MANAGER *manager_resource_hotkey (G_RESOURCE_DESCR *descr, G_LIST *rclist)
{
  MANAGER *self;
  int error = FALSE, used = FALSE;

  self = (MANAGER *) START_METHOD (G_INSTANCE);
  ASSURE (self, "", END_METHOD (NULL));

  /* special command: help prints all possible values */
  if (!strcmp (descr->ident, "help")) {
    used = TRUE;
    printf ("Help for section \"/Manager/hotkey\":\n"
            "\tpossible tags:\n"
            "\tkey string\n"
            "\titem string\n"
            "\tsend string\n"
            "\tclick string\n");
    [...]
    printf ("\thelp\n");
  }

  /* none of them: hand-code the rest */
  if (!error && !used)
    switch (descr->type) {
    case dfInt:
      [...]
      break;
    case dfString:
      if (!strcmp (descr->ident, "key")) {
        GRAPE (self, "add-hotkey")(descr->value);
        used = TRUE;
        hotkey_has_key = TRUE;
      } else if (hotkey_has_key) {
        if (!strcmp (descr->ident, "send")) {
          GRAPE (self, "add-hot-action")(hatSendMethod, descr->value, 0, 1);
          used = TRUE;
        } else if (!strcmp (descr->ident, "click")) {
          GRAPE (self, "add-hot-action")(hatMouseClick, descr->value, 3, 1);
          used = TRUE;
        } [...]
      break;
    default:
      break;
    }

  if (error || !used) {
    g_resourcedev_not_found ((INSTANCE *)self, descr);
    END_METHOD (NULL);
  }
  END_METHOD (self);
}

MANAGER *manager_resource (G_RESOURCE_DESCR *descr, G_LIST *rclist)
{
  MANAGER *self;
  int error = FALSE, used = FALSE, pass = FALSE;

  self = (MANAGER *) START_METHOD (G_INSTANCE);
```

```
  ASSURE (self, "", END_METHOD (NULL));

  /* special command: help prints all possible values */
  if (!strcmp (descr->ident, "help")) {
    used = TRUE;
    pass = TRUE;
    printf ("Help for section \"/Manager\":\n"
            "The simple values are:\n");
    g_resourcelist_print (rclist);
    printf ("Additionally available:\n"
            "\thotkey of type Group\n"
            "\thelp\n");
  }

  /* first try to resolve simple single values */
  if (!error && !used)
    used = g_resourcelist_find (rclist, descr, &error);

  /* none of them: hand-code the rest */
  if (!error && !used)
    switch (descr->type) {
    case dfGroup:
      if (!strcmp (descr->ident, "hotkey")) {
        RESOURCEDEV *rcdev;

        rcdev = (RESOURCEDEV *) GRAPE (ResourceDev, "get-stddev")();
        hotkey_has_key = FALSE;
        GRAPE (rcdev, "parse")("=", self, "resource-hotkey", NULL);
        used = TRUE;
      }
      break;
    default:
      break;
    }

  if ((pass || (!error && !used)) &&
      GRAPE (self, "find-method")("^resource"))
    GRAPE (self, "^resource")(descr, NULL);

  if (error || !used) {
    g_resourcedev_not_found ((INSTANCE *)self, descr);
    END_METHOD (NULL);
  }
  END_METHOD (self);
}

G_LIST *manager_get_rclist (void)
{
  MANAGER *self;
  G_LIST *rclist = NULL;

  self = (MANAGER *) START_METHOD (G_INSTANCE);
  ASSURE (self, "", END_METHOD (NULL));

  if (GRAPE (self, "find-method")("^get-rclist"))
    rclist = (G_LIST *)GRAPE (self, "^get-rclist")();

  rclist = g_rclist_add (rclist, "ask_on_exit", &self->ask_on_exit, dfInt);
```

```
  END_METHOD (rclist);
}

[...]
{
  [...]
  g_resourcedev_parse ((INSTANCE *)mgr, "/Manager");
  [...]
}
```

Chapter 10

Utility Classes and Functions

10.1 Utility Classes

10.1.1 Linked Lists

The class `G_List` implements a doubly linked list of arbitrary objects. The `G_LIST` instance is the list header, it only contains pointers to the first and the last node of the list. The data, pointers to the objects, is store in `G_NODE` structures:

```
typedef struct G_Node {
  struct G_Node *next;   /* next node */
  struct G_Node *prev;   /* previous node */
  void *obj;             /* object at this node */
} G_NODE;

#CLASS G_List : Root {
  char *type;
  G_NODE *head;   /* first element */
  G_NODE *tail;   /* last element */
  G_NODE *iter;   /* current element for iteration functons */
  G_NODE **iter_stack;                    /* stack for nested loops */
  int number_of_iter, max_number_of_iter; /* stack properties       */
};
```

You will not need to access any of the structures directly, doing so can cause incompatibility with further versions of **GRAPE**.

We decided to use structures for the nodes instead of instances because it is much more efficient. For the same reason all operations which work on single nodes are implemented as functions instead of methods.

`G_List`s are used for several internal GRAPE lists, for example all information about classes, methods and items in a project is stored in this way.

10.1.1.1 Basic List Methods and Functions

GRAPE(G_List, "new-instance")(name)

CLASS *G_List
char *name

Create an empty `G_List` instance named `name`. Nodes are created by the insert functions as required.

GRAPE(glist, "free")()

G_LIST *glist

Free all nodes and delete the list. The objects are not freed, therefore any object that is no longer needed should be removed from the list and freed before `"delete"` is called (see insert and remove functions below for an example).

GRAPE(glist, "append")(alist)

G_LIST *glist, *alist

Append the nodes of `alist` to `glist`. The nodes are removed from `alist` which will be empty afterwards.

GRAPE(glist, "remove-all")()

G_LIST *glist

Remove all nodes from `glist` which will be empty afterwards. As `GRAPE(glist, free")()` this method will not free the objects.

GRAPE(glist, "apply")(function)

GRAPE(glist, "apply-method")(method)

G_LIST *glist
void (*function)(void *obj)
char *method

Apply a function or a method to all objects of the list. The function is called with the object as (the only) parameter, for `"apply-method"` it is assumed that the objects are instances, the (parameterless) method is called on them. For example if all instances in a list should be displayed this can be done with

```
GRAPE(glist, "apply-method")("display");
```

If you need to apply some more complex operation to all list objects you should use the functions `g_list_first`, `g_list_last` and `g_list_next`, `g_list_previous` (see below).

GRAPE(glist, "is-empty")()

GRAPE(glist, "contains")(obj)

G_LIST *glist
void *obj

Test if the list is empty or if it contains the given object.

int g_list_add_head(glist, obj)

int g_list_add_tail(glist, obj)

int g_list_add_pre(glist, obj)

int g_list_add_post(glist, obj)

int g_list_insert(glist, obj, prev)

G_LIST *glist
void *obj, *prev

Insert an object into the list. `g_list_add_head` inserts the object as the first and `g_list_add_tail` as the last element, `g_list_insert` inserts it after the given `prev` object, `g_list_add_pre` and `g_list_add_post` before and behind the current position respectively. The functions return `FALSE` if an error occurs, in this case it was not possible to allocate the memory for the node or the object `prev` was not found in the list.

Because the objects of `G_Lists` are of arbitrary type no reference counting can be done by these functions — even if the objects are instances. Therefore you have to be carefull when you insert instances in a list, the reference counter should be incremented when necessary:

```
if(g_list_add_head(glist, inst))
  inst->refcount++;
else
  fprintf(stderr, "can't insert instance %s\n", inst->name);
```

In this case the instances have to be deleted from the list before the list itself is deleted:

```
GRAPE(glist, "apply-method")("delete");
GRAPE(glist, "delete")();
```

void *g_list_rem_head(glist)

void *g_list_rem_tail(glist)

void *g_list_rem_curr(glist)

void *g_list_remove(glist, obj)

G_LIST *glist
void *obj

Remove an object from the list. `g_list_rem_head`, `g_list_rem_tail` and `g_list_rem_curr` remove the first resp. last resp. current element from the list and return it, a return value of `NULL` indicates that the list is empty. `g_list_remove` tries to remove the given object from the list, a return value of `NULL` indicates that the object was not found in the list.

The `g_list_rem_head` or `g_list_rem_tail` function can for example be used to clear a list. If the list contains items which were assigned to the list and added to the manager the list can be cleared with

```
while(inst = g_list_rem_head(glist)) {
  GRAPE(mgr, "remove-inter")(inst);
  GRAPE(inst, "delete")();
}
```

void *g_list_first(glist)

void *g_list_last(glist)

void *g_list_next(glist)

void *g_list_previous(glist)

void *g_list_current(glist)

void *g_list_set_current(glist, obj)

G_LIST *glist
void *obj

Iteration functions for walking through the list. The methods "apply" and "apply-method" can only be used to apply parameterless functions or methods to all elements of the list. For more complex operation on the list elements these functions can be used.

Each list has a current object which can be obtained by `g_list_current` and set by `g_list_set_current`. A `NULL` return value of `g_list_current` indicates that no current object was set before or that the current object was removed from the list, `g_list_set_current` returns `NULL` if `obj` was not found in the list.

`g_list_first` and `g_list_last` set the fist resp. last object of the list as current object and return it, `NULL` as return value indicates that the list is empty. `g_list_next` and `g_list_previous` set the next resp. previous object as current object and return it, they return `NULL` if there is no next or previous object, that is the current object was the last resp. first object of the list.

These functions are for example used by the "apply" method to call the given function on all objects:

```
G_LIST *g_list_apply(void (*func)(void *))
{
  G_LIST *self;
  void *obj;

  self = (G_LIST *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  for(obj = g_list_first(self); obj; obj = g_list_next(self))
    func(obj);

  END_METHOD(self);
}
```

int g_list_count_entries(glist)

G_LIST *glist

`g_list_count_entries` returns the number of nodes in the list.

int g_list_push_iter(glist)

int g_list_dup_iter(glist, depth)

int g_list_xchg_iter(glist, depth)

int g_list_pop_iter(glist, depth)

int g_list_get_curr_iter(glist)

void *g_list_first_for(glist, depth)

void *g_list_last_for(glist, depth)

void *g_list_current_for(glist, depth)

void *g_list_next_for(glist, depth)

void *g_list_previous_for(glist, depth)

G_LIST *glist
int depth

Finding an object in a `G_List` is a slow operation. Nevertheless in some situation you need nested loops with two or more iterators for the same list. There are functions for management of an iterator stack, pushed iterators will be updated if the node they are pointing to is deleted by an operation on another level. Only the top iterator allows read/write operations using the normal `G_List`-functions, the others are read only (if not brought onto the top). Note that the stack is currently designed for local operation only, it is not handled by the `"xdr"` method, though not read/written to/from files.

The `depth` argument can be positive or 0, meaning the absolute (non-changing) position on the stack, or negative, meaning the distance from the top, where -1 is the top (default or current) iterator.

`g_list_push_iter`, `g_list_dup_iter`, `g_list_xchg_iter`, `g_list_pop_iter` and `g_list_get_curr_iter` are the stack management, `g_list_push_iter`, `g_list_dup_iter` push the current resp. the given iterator, `g_list_xchg_iter` exchanges the current with a given one, `g_list_pop_iter` pops all entries up to the given one with the special meaning that `depth == 0` is equivalent to `-1` and pops everything but the standard iterator. `g_list_get_curr_iter`, `g_list_push_iter`, `g_list_dup_iter` return the absolute depth of the current iterator, which will always be positive, or `0` to indicate an error. `g_list_pop_iter` and `g_list_xchg_iter` return `0` or a nonzero value in case of an error resp. successful operation.

`g_list_first_for`, `g_list_last_for`, `g_list_current_for`, `g_list_next_for` and `g_list_previous_for` do the same as their form without `_for`, except that they require a stack depth argument.

An example:

```
type *g_list_test (G_LIST *list, int (*remove)(type *, type *, type *))
{
  type *a, *b, *c;
  int depth;

  for (a = g_list_first (list); a; a = g_list_next (list)) {
    depth = g_list_push_iter (list);
    for (b = g_list_first (list); b; b = g_list_next (list)) {
      g_list_push_iter (list);
      for (c = g_list_first (list); c; c = g_list_next (list)) {
        if (remove(a,b,c)) {
          /* unconditionally pop to the "a" */
          g_list_pop_iter (list, depth);
          /* remove "a" from the list */
          g_list_rem_curr (list);
          return a;
        }
      }
      /* pop one stack entry: back to "b" */
      g_list_pop_iter (list, 0);
    }
    g_list_pop_iter (list, 0);
  }
  return NULL;
}
```

10.1.1.2 Methods for Instance Lists

The following methods assume that the objects in the list are instances.

GRAPE(glist, "hardcopy")()

GRAPE(glist, "softcopy")(copy)

GRAPE(glist, "copy")()

G_LIST *glist, *copy

All three methods create a copy of the list and its nodes and return it. The difference is how the instance objects are copied: `"hardcopy"` and `"softcopy"` use the corresponding methods to copy them, `"copy"` only assigns the object of the original list to the copy (the reference counter is incremented). As always `"softcopy"` uses the memory of a given `copy` list to store the copy.

10.1.2 List_Of_Inst

For `int`, `float`, `double` and `VEC3` there exist `List_Of_[type]` classes. Each `List_Of_[type]` class has the entries `flag`, `max_num_of_[type]`, `num_of_[type]` and `[type]*vec`. These classes have no special methods at the moment.

`List_Of_Inst` is almost like the other `List_Of_[type]` classes but instead of `*vec` there is a `INSTANCE **objects` entry. `List_Of_Inst` is for example used to store the boundaries of `Geom2d` instances (6.3.4), and for storing information about the connectivity of triangulations (see ->connectivity methods on `Triang2d` (6.3.1.4)).

```
#CLASS List_Of_Inst : Root {
    int num_of_objects;
    int max_num_of_objects;
    int *flags;
    INSTANCE **objects;
};
```

10.1.2.1 Memory Methods on List_Of_Inst

GRAPE(list, "free")()

`LIST_OF_INST *list`

Delete an instance of class `List_Of_Inst`. All instances of the `list` are deleted by calling `"list-free"`, then the instance itself is deleted.

GRAPE(list, "list_alloc")(length, class, name)GRAPE(list, "list_free")()

```
LIST_OF_INST *list
int length
CLASS *class
char *name
```

`"list_alloc"` allocates the memory for the flag and instance arrays. If `class` is not `NULL` then `length` instances of class `class` with name `name` are allocated. The `flags` for all `objects` is set to `ACTIVE`.

`"list_free"` deletes all `objects` by sending `"delete"` to them and frees the `flags` and `objects` lists.

GRAPE(list, "hardcopy")()

`LIST_OF_INST *list`

Make a hardcopy of `list`: a new `List_Of_Inst` instance with new lists is created, the `flags` are copied and all `objects` are hardcopied. The new list is returned.

GRAPE(list, "softcopy")(copy_list)

```
LIST_OF_INST *list
LIST_OF_INST *copy_list
```

Make a softcopy of `list`: like `"hardcopy"` but the `objects` are softcopied (only the used data is copied). If `copy_list` is not `NULL` then this instance is used to store the copy otherwise a new instance is created.

GRAPE(list, "reflect")(trfmat, orient)

```
LIST_OF_INST *list
MATRIX44 trfmat
int orient
```

Create a new `List_Of_Inst` instance which is created almost like a `"hardcopy"` but instead of sending `"hardcopy"` to each object `"reflect"` is used. The new instance is transformed with `trfmat`, `orient < 0` results in a change of the orientation of the new objects.

(int *)GRAPE(list, "merge")(list2)

```
LIST_OF_INST *list
LIST_OF_INST *list2
```

Merge the two instances `list` and `list2` by appending the lists of `list2` to those of `list`. If anything fails `"merge"` returns `NULL`.

10.1.2.2 Other Methods on List-Of-Inst

GRAPE(list, "activate_send")()GRAPE(list, "deactivate_send")()

```
LIST_OF_INST *list
```

Add/remove the `ACTIVE` bit for all `objects` in the `list`. This flag is used in

GRAPE(list, "universal")(method)

```
LIST_OF_INST *list
char *method
```

Send `method` to all active objects (`ACTIVE` bit set in `flags[i]`).

GRAPE(list, "interpol")(list1, list2, factor)

GRAPE(list, "interpol_fac")(list1, list2, fac1, fac2)

```
LIST_OF_INST *list
LIST_OF_INST *list1, *list2
double factor, fac1, fac2
```

Interpolate the two `List_Of_Inst` instances `list1` and `list2` with `factor`: `list = (1 − factor) * list1 + factor * list2` or with factors `fac1` and `fac2`: `list = fac1 * list1 + fac2 * list2`. If necessary new memory is allocated for `list`.

GRAPE(list, "get_info")(xlist)

```
LIST_OF_INST *list
XLIST *xlist
```

Create info `xlist` entries for a `List_Of_Inst` instance. Entries are class, name, refcount, number of objects and class and flag for each object.

10.2 Utility Functions

GRAPE provides some useful utility functions that are frequently used. Included are not only mathematical utility functions, but also routines for:

- memory management,
- vector calculations (`VEC2`, `VEC3`, `VEC4` and n_dim vectors) and
- matrix operations

Furthermore a number of complex functions that are available in **GRAPE** simplify the handling of complex variables.

10.2.1 Memory Functions

10.2.1.1 Basic Memory Functions

All memory functions are based on the functions `mem_alloc`, `mem_free` and `mem_realloc` which are described in section 5.2.2. There are special memory functions for all normal datatypes and the **GRAPE** datatypes. These functions will be described in the next sections and should be used instead of the general functions.

`void *`**mem_alloc** `(size_t size)`
Allocate a memory block of the size `size*sizeof(char)` and will return the pointer to the block. If the allocation fails the function will return a `NULL`-pointer.

`void` **mem_free** `(void *ptr, size_t size)`
This function will free a block at the adress ptr with the size `size*sizeof(char)`. You can see that the function `mem_free` needs the size of the block, so you need to store the size of a block you allocate to free it correctly. To free a block without knowing its size you can use the special value `G_BLOCKSIZE` instead of the real size, but giving a size enables `mem_free` to check if your program assumed a wrong size.

`void *`**mem_realloc** `(void *ptr, size_t old_size, size_t new_size)`
This function resizes a memory block. It is basically the same as allocating a new block, copying the data, freeing the original block and returning the new one. Though `mem_realloc` tries not to move the block, shrinking will never move it. In case of an error `mem_alloc` returns `NULL` and the old value of `ptr` will still be valid. `G_BLOCKSIZE` is allowed as `old_size`, but not recommended, as in `mem_free`. If `ptr` is `NULL`, `mem_realloc` does the same as `mem_alloc`, if `new_size` is zero the block is simply freed.

10.2.1.2 Fast Functions for Large Numbers of Small Blocks

Many applications need lots of equally sized small memory blocks which have to be allocated and freed very fast.

Normally one would write some sort of pointer arithmetic or free list management, one for each structure, which splits superblocks into many small blocks. For simplicity these functions will not free any memory if not everything at a time, sometimes the capacity is limited.

The easy way is to use the **GRAPE** utility functions designed for this purpose. The following functions are a low level application of `mem_alloc/mem_free` implementing a configurable free list manager which will free unused superblocks if possible, at the only penalty of losing at most `sizeof(void *)` bytes per block.

`void *`**g_bmm_new** `(size_t blocksize, size_t alloc_number)`
Create a new block memory manager which is returned, which manages blocks of size `blocksize`, allocating `alloc_number` at a time.

`void` **g_bmm_destroy** `(void *bmm)`
Totally destroy a block memory manager and all its managed blocks.

`int` **g_bmm_is_empty** `(void *bmm)`
`TRUE`, if there are no managed blocks.

`void` ***g_bmm_alloc** `(void *bmm)`
Return a new block.

`void` **g_bmm_free** `(void *bmm, void *ptr)`
Free a block.

`void` **g_bmm_new_num** `(void *bmm, size_t new_alloc_number)`
Change the `alloc_number` of `bmm`, which will be used for all superblock allocations. This does not influence already allocated ones.

`void` **g_bmm_shrink** `(void *bmm)`
If you free some blocks one empty superblock is kept for further allocations, as a sort of cache. Free this one.

You should call this function after having done a lot of memory operations.

`void` ***g_bmm_get_std** `(size_t blocksize)`
Return the system wide standarb block memory manager for this blocksize. Of course it cannot be destroyed using `g_bmm_destroy`, you have to call `g_bmm_free` for any block. Though using the standard manager saves memory, because there are not so many half used superblocks.

An example using a private block memory manager, all blocks are freed with one function call at the end:

```
{
  void *bmm;
  struct my_struct *p[1000];
  int i;

  bmm = g_bmm_new (sizeof (my_struct), 100);

  /* use manager */
  for (i = 0; i < 1000; i++)
    p[i] = g_bmm_alloc (bmm);
  for (i = 0; i < 100; i++)
    g_bmm_free (bmm, p[i*10]);
  for (i = 0; i < 10; i++)
    p[i*100] = g_bmm_alloc (bmm);
  ...
  g_bmm_destroy (bmm);
}
```

An example using the system wide standard block memory manager for this blocksize:

```
{
  void *bmm;
  struct my_struct *p[1000];
  int i;

  bmm = g_bmm_get_std (sizeof (my_struct));
```

```
    /* use manager */
    for (i = 0; i < 1000; i++)
      p[i] = g_bmm_alloc (bmm);
    ...
    for (i = 0; i < 1000; i++)
      g_bmm_free (bmm, p[i]);
  }
```

10.2.1.3 Functions for allocating/freeing memory of data-arrays

```
char *char_alloc(size_t nr_of_char)
char *char_free(char *ptr, size_t nr_of_char)
int *int_alloc(size_t nr_of_int)
int *int_free(int *ptr, size_t nr_of_int)
double *double_alloc(size_t nr_of_double)
double *double_free(double *ptr, size_t nr_of_double)
```

All this allocate functions will return a pointer to the allocated array which is big enough fo nr_of_...elements of the specific type. If the function fails it will return `NULL`. The free functions will free a memory block which is big enough for nr_of_...elements at the address `ptr` points to. The free functions will return the `NULL` pointer.

10.2.1.4 Functions for allocating/freeing memory of pointer-arrays

```
double **double_p_alloc(size_t nr_of_double_p)
double **double_p_free(double **ptr, size_t nr_of_double_p)
char **char_p_alloc(size_t nr_of_char_p)
char **char_p_free(char **ptr, size_t nr_of_char_p)
```

These allocation functions will allocate memory for an array of pointer of the specific type. The array will have place for nr_of_... pointer. These functions will return a NULL-pointer if alloction fails, too. The free functions work as the one described above. Pointer arrays are initialized with `NULL` pointers by the alloc functions.

10.2.1.5 Functions for allocating/freeing memory of arrays of GRAPE datatypes

We will not list the memory functions of all **GRAPE** datatypes here, but give only some examples. These functions exist for all **GRAPE** datatypes.

```
INT2 *int2_alloc(size_t nr_of_int2)
INT2 *int2_free(INT2 *ptr, size_t nr_of_int2)
INT2 **int2_p_alloc(size_t nr_of_int2_p)
INT2 **int2_p_free(INT2 **ptr, size_t nr_of_int2_p)
VEC3 *vec3_alloc(size_t nr_of_vec3)
VEC3 *vec3_free(VEC3 *ptr, size_t nr_of_vec3)
VEC3 **vec3_p_alloc(size_t nr_of_vec3_p)
VEC3 **vec3_p_free(VEC3 **ptr, size_t nr_of_vec3_p)
```

All these functions work exactly like ones described in the last two section. The names of the identic functions for the other **GRAPE** datatypes are:

```
datatype_alloc(size_t nr_of_datatype)
datatype_free(datatype *ptr, size_t nr_of_datatype)
datatype_realloc(datatype *ptr, size_t old_nr_of_datatype, \
          size_t new_nr_of_datatype)
g_smart_datatype_realloc(datatype *ptr, size_t nr_of_datatype, \
```

```
        size_t old_max_nr_of_datatype, size_t new_max_nr_of_datatype)
datatype_p_alloc(size_t nr_of_datatype_p)
datatype_p_free(datatype **ptr, size_t nr_of_datatype_p)
```

The `_realloc`-functions change a block's size and return the maybe changed address, data contained in both blocks is preserved. To save copy time there is a special version of some of the `_realloc`-functions, called `g_smart_datatype_realloc`, based on the generic function `g_smart_mem_realloc`, where you pass the amount of used memory that has to be copied if necessary. In addition these functions check their arguments, complaining about more used then previously allocated memory or more used then new allocated memory. Sometimes only the amount of used memory has been wrongly set, as in this example:

```
{
  TRIANG2D *t;
  int i;
  t = (TRIANG2D *)GRAPE (Triang2d, "new-instance")("my object");
  t->number_of_points = 10;
  GRAPE (t, "list-alloc")(10, 10);
  ... /* fill arrays*/
}
```

Since `"list-alloc"` internally uses `g_smart_double_realloc` you will get this message for the `x`, `y` and `z` arrays of `t`:

```
g_smart_mem_realloc: number > max_number
        Active Method : 'list-alloc' (Triang2d) [0]
               Object : "my object" (Triang2d)
```

Set `number_of_points` after allocating the lists, which is the correct usage anyway.

A complete list of all existing functions:

```
void *g_smart_mem_realloc
double *double_alloc, double_free, double_realloc, g_smart_double_realloc
double **double_p_alloc, double_p_free, double_p_realloc
char *char_alloc, char_free, char_realloc, g_smart_char_realloc
char **char_p_alloc, char_p_free, char_p_realloc
int *int_alloc, int_free, int_realloc, g_smart_int_realloc
int **int_p_alloc, int_p_free, int_p_realloc
INT2 *int2_alloc, int2_free, int2_realloc, g_smart_int2_realloc
INT2 **int2_p_alloc, int2_p_free, int2_p_realloc
INT3 *int3_alloc, int3_free, int3_realloc, g_smart_int3_realloc
INT3 **int3_p_alloc, int3_p_free, int3_p_realloc
INT4 *int4_alloc, int4_free, int4_realloc, g_smart_int4_realloc
INT4 **int4_p_alloc, int4_p_free, int4_p_realloc
VEC2 *vec2_alloc, vec2_free, vec2_realloc, g_smart_vec2_realloc
VEC2 **vec2_p_alloc, vec2_p_free, vec2_p_realloc
VEC3 *vec3_alloc, vec3_free, vec3_realloc, g_smart_vec3_realloc
VEC3 **vec3_p_alloc, vec3_p_free, vec3_p_realloc
VEC4 *vec4_alloc, vec4_free, vec4_realloc, g_smart_vec4_realloc
VEC4 **vec4_p_alloc, vec4_p_free, vec4_p_realloc
INSTANCE **instance_p_alloc, instance_p_free, instance_p_realloc
CHAIN **chain_p_alloc, chain_p_free, chain_p_realloc
SPLINE1D **spline1d_p_alloc, spline1d_p_free, spline1d_p_realloc
TRIANG1D **triang1d_p_alloc, triang1d_p_free, triang1d_p_realloc
TRIANG2D **triang2d_p_alloc, triang2d_p_free, triang2d_p_realloc
TRIANG1D ***triang1d_pp_alloc, triang1d_pp_free, triang1d_pp_realloc
TRIANG2D ***triang2d_pp_alloc, triang2d_pp_free, triang2d_pp_realloc
TRIANG1D ****triang1d_ppp_alloc, triang1d_ppp_free, triang1d_ppp_realloc
TRIANG2D ****triang2d_ppp_alloc, triang2d_ppp_free, triang2d_ppp_realloc
```

```
FUNCTION_RULER **function_ruler_p_alloc, function_ruler_p_free,
                 function_ruler_p_realloc
BOUNDED_RULER **bounded_ruler_p_alloc, bounded_ruler_p_free,
                bounded_ruler_p_realloc
RULER **ruler_p_alloc, ruler_p_free, ruler_p_realloc
```

10.2.2 Functions on VEC2

`VEC2` is declared as **typedef double VEC2[2];** and the following functions are available:

`double` **g_abs2** `(double a, double b)`
: `g_sqrt`(`a`*`a` + `b`*`b`)

`double` **g_vec2_abs** `(VEC2 v)`
: returns the length of `v`

`void` **g_vec2_add** `(VEC2 u, VEC2 v, VEC2 w)`
: `u` = `v` + `w`

`double` **g_vec2_angle** `(VEC2 v,VEC2 w)`
: returns the signed angle of `v` and `w`

`void` **g_vec2_assign** `(VEC2 v, VEC2 w)`
: `v` = `w`

`void` **g_vec2_dec** `(VEC2 v, VEC2 w)`
: `v` -= `w`

`double` **g_vec2_det** `(VEC2 v,VEC2 w)`
: returns `v[0]*w[1]-v[1]*w[0]`

`double` **g_vec2_dist** `(VEC2 v,VEC2 w)`
: returns the distance of the two points in R2

`void` **g_vec2_inc** `(VEC2 v, VEC2 w)`
: `v` += `w`

`void` **g_vec2_mult** `(VEC2 v,double d)`
: multiplies `v` by the number `r`

`void` **g_vec2_rot90** `(VEC2 v, VEC2 w)`
: `v` = `w` rotated by 90 degree

`void` **g_vec2_set** `(VEC2 v, double a, double b)`
: `v` = `(a,b)`

`void` **g_vec2_set_zero** `(VEC2 v)`
: `v` = `(0,0)`

`double` **g_vec2_skp** `(VEC2 v,VEC2 w)`
: returns the scalar product of `v` and `w`

`double` **g_vec2_sqrabs** `(VEC2 v)`
: returns the square length of `v`

`double` **g_vec2_sqrdist** `(VEC2 v,VEC2 w)`
: returns the square distance of two points

`void` **g_vec2_sub** `(VEC2 u, VEC2 v, VEC2 w)`
: `u` = `v` - `w`

`double` **g_vec2_sub_skp** `(VEC2 v,VEC2 w,VEC2 n)`
: returns the scalar product of `v` - `w` and `n`

10.2.3 Functions on VEC3

`VEC3` is declared as **typedef double VEC3[3];** and the following functions are available:

`void` **g_matrix44_mult_vec3** `(MATRIX44 m, VEC3 v)`
multiplies `v` with the matrix `m`

`void` **g_matrix44_proj_vec3** `(VEC3 u, MATRIX44 m, VEC3 v)`
multiplies `(v,1)` with the matrix `m`, divide the first three components of the result by the fourth and return them in `u`

`double` **g_vec3_abs** `(VEC3 v)`
returns the length of `v`

`void` **g_vec3_add** `(VEC3 vadd, VEC3 v, VEC3 w)`
`vadd` = `v` + `w`

`double` **g_vec3_angle (VEC3 p, VEC3 q1, VEC3 q2)**
returns the angle at `p` in the triangle `q1_p_q2`

`void` **g_vec3_assign** `(VEC3 v, VEC3 w)`
`v` = `w`

`int` **g_vec3_assign_triang_node** `(VEC3 v, TRIANG2D *t, int n)`
copies the nth point of `t` to `v`

`double` **g_vec3_circle** `(VEC3 center, VEC3 p[3])`
calculates the center of the circle through `p[0]`, `p[1]` and `p[2]` and returns the radius

`int` **g_vec3_cmp** `(VEC3 a, VEC3 b, double eps)`
returns 1 if the square distance of `a` and `b` is smaller than `eps`

`void` **g_vec3_crossprod** `(VEC3 u, VEC3 v, VEC3 w)`
`u` = cross product of `v` and `w`

`double` **g_vec3_ctg** `(VEC3 v, VEC3 w)`
returns the cotangens of the angle between `v` and `w`

`void` **g_vec3_dec** `(VEC3 v, VEC3 w)`
`v` -= `w`

`double` **g_vec3_dist** `(VEC3 p, VEC3 q)`
returns the distance of the two points in R3

`int` **g_vec3_get_arbitr_normal** `(VEC3 normal, VEC3 vec)`
`normal` = a unit vector orthogonal on `vec` returns 0 in case of an error

`int` **g_vec3_get_normal** `(VEC3 u, VEC3 v, VEC3 w)`
`u` = a unit vector orthogonal on `v` and `w` returns 0 in case of an error

`int` **g_vec3_get_normal_to_plane** `(VEC3 u, VEC3 p, VEC3 q, VEC3 r)`
`u` = the normal at `p` of the triangle `p_q_r`

`void` **g_vec3_inc** `(VEC3 v, VEC3 w)`
`v` += `w`

`void` **g_vec3_invert** `(VEC3 v)`
`v = (1,1,1) - v`

`void` **g_vec3_list_assign** `(VEC3 v, VEC3 w, int nelem)`
copies the vector list `w` to `v`

`void` **g_vec3_mult** `(VEC3 v, double r)`
multiplies `v` by the number `r`

`void` **g_vec3_normalize** `(VEC3 v)`
calculates a unit vector from `v`, replacing `v`, returns 0 in case of an error

`int` **g_vec3_proj_onto_line** `(VEC3 v, VEC3 u)`
replaces `v` by its projection on span (`u`) (if `u` has unit length)

`int` **g_vec3_proj_onto_plane** `(VEC3 v, VEC3 u)`
replaces `v` by its projection on the plane with normal `u`

`void` **g_vec3_set** `(VEC3 v, double v0, double v1, double v2)`
`v` = `(v0, v1, v2)`

`void` **g_vec3_set_length** `(VEC3 v, double r)`
calculates a vector of length `r` from `v`, replacing `v`

`void` **g_vec3_set_zero** `(VEC3 v)`
`v` = `(0, 0, 0)`

`double` **g_vec3_skp** `(VEC3 v, VEC3 w)`
returns the scalar product of `v` and `w`

`double` **g_vec3_sqrabs** `(VEC3 v)`
returns the square length of `v`

`double` **g_vec3_sqrdist** `(VEC3 p, VEC3 q)`
returns the square distance of two points

`void` **g_vec3_sub** `(VEC3 vsub, VEC3 v, VEC3 w)`
`vsub` = `v - w`

`void` **g_fvec3_cv_vec3** `(VEC3 v, FVEC3 w)`

`void` **g_vec3_cv_fvec3** `(FVEC3 v, VEC3 w)`
`v` = `w`, where `typedef float FVEC3[3];`

10.2.4 Functions on VEC4

`VEC4` is declared as **typedef double VEC4[4];**
Some of the functions for 4-dimensional vectors depend on signature. Signature is a `VEC4` vector describing a metric, e.g.:

```
euclidian R4:  ( 1., 1., 1., 1.)
minkowski R4:  (-1., 1., 1., 1.)
```

The following functions are available in **GRAPE**:

`void` **g_matrix44_mult_vec4** `(MATRIX44 m, VEC4 v)`
multiplies `v` by the matrix `m`, replacing `v`

`void` **g_matrix44_mult_vec4_vec4** `(MATRIX44 m, VEC4 v, VEC4 w)`
multiplies `v` by `m` and stores the result in `w`

`double` **g_vec4_abs** `(VEC4 v)`
returns the length of `v`

`double` **g_vec4_abs_sign** `(VEC4 v, VEC4 sign)`
returns the length of `v` (depending on signature)

`void` **g_vec4_add** `(VEC4 vadd, VEC4 v, VEC4 w)`
  `vadd` = `v` + `w`

`double` **g_vec4_angle** `(VEC4 p, VEC4 q1, VEC4 q2, VEC4 sign)`
  returns the angle at `p` in the triangle `q1-p-q2` (depending on signature)

`void` **g_vec4_assign** `(VEC4 v, VEC4 w)`
  `v` = `w`

`double` **g_vec4_ctg** `(VEC4 p, VEC4 q1, VEC4 q2, VEC4 sign)`
  returns the cotangens of the angle at p in the triangle `q1-p-q2` (depending on signature)

`double` **g_vec4_dist** `(VEC4 p, VEC4 q)`
  returns the distance of the two points

`double` **g_vec4_dist_sign** `(VEC4 p, VEC4 q, VEC4 sign)`
  returns the distance of two points (depending on signature)

`void` **g_vec4_list_assign** `(VEC4 *v, VEC4 *w, int nelem)`
  copies the list of vectors `w` to the list `v`

`void` **g_vec4_mult** `(VEC4 v, double r)`
  multiplies each coordinate of `v` by `r`, replacing `v`

`int` **g_vec4_normalize** `(VEC4 v)`
  creates a unit vector from `v`, replacing `v`

`int` **g_vec4_normalize_sign** `(VEC4 v, VEC4 sign)`
  creates a unit vector from `v` (depending on signature)

`int` **g_vec4_set_length** `(VEC4 v, double r)`
  creates a vector of length `r` from `v`, replacing `v`

`int` **g_vec4_set_length_sign** `(VEC4 v, double r, VEC4 sign)`
  creates a vector of length `r` from `v` (depending on signature)

`double` **g_vec4_skp** `(VEC4 v, VEC4 w)`
  returns the scalar product of `v` with `w`

`double` **g_vec4_skp_sign** `(VEC4 v, VEC4 w, VEC4 sign)`
  returns the scalar product of `v` and `w` (depending on signature)

`double` **g_vec4_sqrabs** `(VEC4 v)`
  returns the square of the length of `v`

`double` **g_vec4_sqrabs_sign** `(VEC4 v, VEC4 sign)`
  returns the square length of `v` (depending on signature)

`double` **g_vec4_sqrdist** `(VEC4 p, VEC4 q)`
  returns the square of the distance of the two points

`double` **g_vec4_sqrdist_sign** `(VEC4 p, VEC4 q, VEC4 sign)`
  returns the square distance of two points (depending on signature)

`void` **g_vec4_sub** `(VEC4 vsub, VEC4 v, VEC4 w)`
  `vsub` = `v` - `w`

10.2.5 Matrix Functions

Matrices are declared as double arrays: **typedef double MATRIX44[4][4];**

The following matrix-operations are possible:

`void` **g_matrix44_assign** `(MATRIX44 s, MATRIX44 d)`
: `d` = `s`

`double` **g_matrix44_det** `(MATRIX44 m)`
: returns the determinant of the main 3x3 part of `m`

`int` **g_matrix44_inv** `(MATRIX44 m, MATRIX44 d)`
: inverts the main 3x3 part of `m` and stores result in `d`

`int` **g_matrix44_invert_gauss** `(MATRIX44 m, MATRIX44 d)`
: inverts `m` and stores result in `d`

`void` **g_matrix44_mult** `(MATRIX44 r, MATRIX44 m1, MATRIX44 m2)`
: `r` = `m1` * `m2`

`void` **g_matrix44_mult_vec3** `(MATRIX44 m, VEC3 v)`
: `v` = `m` * `v`

`void` **g_matrix44_mult_vec4** `(MATRIX44 m, VEC4 v)`
: `v` = `m` * `v`, where `v` is of the type `VEC3` or `VEC4`

`void` **g_matrix44_mult_vec4_vec4** `(MATRIX44 m, VEC4 v, VEC4 w)`
: `w` = `m` * `v`

`void` **g_matrix44_proj_vec3** `(VEC3 u, MATRIX44 m, VEC3 v)`
: multiply `(v,1)` with the matrix `m`, divide the first three components of the result by the fourth and return them in `u`

`void` **g_matrix44_set_identity** `(MATRIX44 m)`
: `m` = identity matrix

`void` **g_matrix44_transp** `(MATRIX44 m, MATRIX44 t)`
: transposes `m` and stores result in `t`

`int` **g_solve2** `(double a[2][2],VEC2 b,VEC2 x)`

`int` **g_solve3** `(MATRIX33 a,VEC3 b,VEC3 x)`

`int` **g_solve4** `(MATRIX44 a,VEC4 b,VEC4 x)`
: find `x` with `a`*`x` = `b`, return 1 of ok. `g_solve3` is using a slow algorithm.

10.2.6 Functions for Arrays and n-dim Vectors

`int` **g_get_vecpos_int** `(int x, int *v, int n)`
: searches `x` in `v[n]`; returns the index, if found, otherwise -1

`void` **g_int3_assign** `(INT3 v, INT3 w)`
: `v` = `w`

`void` **g_int3_list_assign** `(INT3 *v, INT3 *w, int n)`
: copy the `INT3` list `w` of length `n` to `v`

`void` **g_get_vecpos** `(double *v, int n, double x, int *i,double *fraction)`
: searches for `x` in `v[n]` and stores the resulting index in `i`

double ***g_vec_append_double** (double **v1, int *n1, int *m, double *v2, int n2)

float ***g_vec_append_float** (float **v1, int *n1, int *m, float *v2, int n2)

INSTANCE ****g_vec_append_instance** (INSTANCE ***v1, int *n1, int *m, INSTANCE **v2, int n2)

int ***g_vec_append_int** (int **v1, int *n1, int *m, int *v2, int n2)

INT3 ***g_vec_append_int3** (INT3 **v1, int *n1, int *m, INT3 *v2, int n2)

VEC3 ***g_vec_append_vec3** (VEC3 **v1, int *n1, int *m, VEC3 *v2, int n2)
append `v2[n2]` to `v1[n1]`; if `n1` + `n2` > `m`, then allocate new `v1` and free old one. The elements of `v1` and `v2` have the same type as the function.

void **g_vec_change_int** (int *v, int n, int *v1, int *v2, int m)

void **g_vec_change_int3** (INT3 *v, int n, int *v1, int *v2, int m)
change in `v[n]` all occurrences of numbers in `v1[m]` to corresponding numbers in `v2[m]`

void **g_vec_change_value_int** (int *v, int n, int x1, int x2)

void **g_vec_change_value_int3** (INT3 *v, int n, int x1, int x2)
change in `v[n]` all values `x1` to `x2`

int **g_vec_cmp_float** (float *a, float *b, int n, double eps)
returns 1 if the `n`-dim vectors `a` and `b` have square distance less than `eps`

double **g_vec_max_double** (double *v, int n)

float **g_vec_max_float** (float *v, int n)

int **g_vec_max_int** (int *v, int n)
return the maximum entry in the `n`-dim array `v`

double **g_vec_min_double** (double *v, int n)

float **g_vec_min_float** (float *v, int n)

int **g_vec_min_int** (int v, int n)
return the minimum entry in the `n`-dim array `v`

10.2.7 Transformation Functions

The transformation functions take some arguments that describe a transformation (`reflect`, `rotate with angle phi` ...) in 3dim-space and then return the transformation's matrix representation `m`. A line is given by two points (`p[2]`), a plane by three points (`p[3]`).

int **g_line_reflect** (VEC3 p[2], MATRIX44 m)

int **g_line_rotate** (VEC3 p[2], double phi, MATRIX44 m)

int **g_plane_reflect** (VEC3 p[3], MATRIX44 m)

int **g_point_reflect** (VEC3 p, MATRIX44 m)

int **g_translate** (VEC3 p, MATRIX44 m)

10.2.8 Function Interpreter Functions

GRAPE contains a simple but easy to use function interpreter. Once you have a function string (e.g. `"x*x+sqrt(y)-z*t"`) you have to precompile it to a `t_fun`, providing the variable names. This `t_fun` can be executed fast enough for most purposes.

The interpreter only knows about scalar `double` values, it recognizes the operators and functions +, -, *, /, ^, (), `arccos`, `arcsin`, `arctan`, `cos`, `cosh`, `exp`, `log`, `sin`, `sinh`, `sqrt`, `tan`, `tanh` and the constant `pi`.

`t_fun` ***fn_func** `(char *in, char *varnames)`
: Precompile the function `in`. The variable names have to be in order ans space separated in `varnames`, e.g. `"x y z t"`. In case of an error `NULL` is returned; use `fn_errstr` and `fn_errpos` for diagnostics.

`t_fun` ***fn_delete** `(t_fun *fn)`
: Free the compiled code, return `NULL`.

`double` **fn_eval** `(t_fun *fn, double *vars)`
: Evaluate the compiled function code `fn`. `vars` has to contain the variable values in order, e.g. `vars[0]=x,vars[1]=y,vars[2]=z,vars[3]=g_maintime`. The result value is returned.

`char` ***fn_errstr** `()`
: Return an error message in case `fn_eval` returned `NULL`.

`char` ***fn_errpos** `()`
: Return the error position in the function string in case `fn_eval` returned `NULL`.

10.2.9 String Functions

The **GRAPE** memory manager prohibits to use standard string functions which allocate memory using `malloc`. For this reason some utility functions were designed:

`char` ***g_strdup** `(const char *source)`
: Do the same as `strdup`, but using `char_alloc`: Allocate `strlen(source)+1` chars, copy the given string there and return it.

`char` ***g_strfree** `(char *str)`
: Free `strlen(str)+1` chars, return `NULL`. An easy way to free ("packed") strings, e.g. allocated with `g_strdup`.

`char` ***g_strchange** `(char *dest, const char *source)`
: Easy way to change a ("packed") string, e.g. allocated with `g_strdup`. Free `dest` and then allocate and return a copy of `source`.

`char` ***g_strcat** `(char *a, const char *b)`
: Return `a = strcat (a, b)`. `a` is reallocated and se t the concatenated `a` and `b`, `b` is not changed. That means: `a` has to be a `g_strdup`-style string or `NULL`, `b` can be any string or `NULL`.

`char` ***g_strcut** `(char *str, size_t len)`
: Cuts `g_strdup`-style string `str` to have `<= len` characters, i.e. `<= len + 1` characters including terminating `0`.

`int` **g_strempty** `(char *str)`
: Returns `FALSE` if `str` equals `NULL` or the empty string, otherwise `TRUE`.

10.2.10 Method String Functions

The **GRAPE** method dispatcher is much faster if the method string contains a unique numeral code. On the other hand method strings are much more readable if not containing this code and `strcmp` will not work with the code part. The code will vary at every program start, so do not save enhanced method strings in files.

This is useful for method strings that are stored in variables or passed as function arguments. Because a constant string's method code can be found by hashing its address and constant function arguments will never change, there is no need to apply this functions on simple method call's arguments like that (constant arguments are easily identified by their leading "`"`"):

```
GRAPE (obj, "my-method")();
```

Of course the underlying concepts can change with new versions of **GRAPE**.

`char` ***g_convert_method*** `(char const * const method)`
Allocate memory for the enhanced method string, calculate the method code, concatenate the passed string and return the resulting enhanced method string. You have to free the result using `g_strfree`. Already enhanced strings are simply `g_strdup`ped, `NULL` will be returned unchanged.

If an unknown string is detected, the function generates a new number and stores a copy of `method`, so to save memory be careful not to pass random strings.

`char` ***g_strip_method*** `(char * const method)`
Return the address of the first character of the readable string in `method`. Since the code is added at the head you get the original string. This function detects normal strings or `NULL` strings and returns the argument in this cases.

10.2.11 Swap Functions

To exchange the two variables `a` and `b` you can use the following swap functions:

`void` **g_swap_int** `(int *a, int *b)`

`void` **g_swap_float** `(float *a, float *b)`

`void` **g_swap_double** `(double *a, double *b)`

`void` **g_swap_vec3** `(VEC3 a, VEC3 b)`

`void` **g_swap_vec4** `(VEC4 a, VEC4 b)`

10.2.12 Drawing Routines

`void` **g_draw_circle** `(VEC3 midpoint, double radius, VEC3 normal, int n, GRAPHICDEVICE *dev)`
draws a regular polygon outline with `n` vertices in the plane orthogonal to `normal`

`int` **g_draw_cyl** `(VEC3 *base_circle, VEC3 *head_circle, VEC3 *base_norm, VEC3 *head_norm, int n, int orient, int color_flag, VEC3 color, GRAPHICDEVICE *dev)`
draws a cylinder with base circle given as a `n`-dim array `base_circle` and head circle given as a `n`-dim array `head_circle`

`int` **g_draw_arrow** `(VEC3 base, VEC3 head, double thickness, /* default == 1 */ int n, int orient, int color_flag, VEC3 color, GRAPHICDEVICE *dev)`
draws an arrow with base point `base` and head point `head` with `n` vertices

`int` **g_draw_tube** `(INSTANCE *obj, int nop, int (*getpoint)(VEC3 v, INSTANCE *obj, int i), int discr, double thickness)`
draws a tube around `obj`, which is assumed to have vertices in a given order. `nop` is the number of these vertices, `getpoint` has to fill its `v` argument with the `i`-th point of `obj` and return non-0, or to return 0 in case of an error. `discr` is the number of points of each tube segment; if `discr` is 1, a line is drawn instead. `thickness` is a linear scaling factor for the tube radius.

10.2.13 Time Functions

Since time dependent functions — especially with short time intervals — are machine dependent, we implemented some ones of general interest. Note that the values will wrap around every 71 minutes.

`unsigned long` **g_time_usec** `(void)`
Returns the real time in microseconds.

`unsigned long` **g_usertime_usec** `(void)`
Returns the user time in microseconds, not including the time for system calls.

`void` **g_sleep_usec** `(unsigned long)`
Sleep the given amount of microseconds.

10.2.14 General Utility Macros

Some macros of general interest:

```
#define G_ABS(A)   (  0  < (A) ?  (A) : (-(A)) )
#define G_MAX(A,B) ( (A) > (B) ?  (A) :   (B)  )
#define G_MIN(A,B) ( (A) < (B) ?  (A) :   (B)  )
#define G_SGN(A)   ( (A) <  0  ? (-1) :    1   )
```

10.2.15 Math Utility Functions

These basic functions are supplied for compatibility reasons because the parameter and return types and even the names of the basic math functions are not standardized. There exists single and double precision versions of these functions, they are implemented as macros if possible.

`double` **g_sqrt** `(double x)`

`float` **g_sqrtf** `(float x)`

`double` **g_fabs** `(double x)`

`float` **g_fabsf** `(float x)`

`double` **g_pow** `(double x, double y)`

`float` **g_powf** `(float x, float y)`

`double` **g_cos** `(double x)`

`float` **g_cosf** `(float x)`

`double` **g_sin** `(double x)`

float **g_sinf** (float x)

double **g_tan** (double x)

float **g_tanf** (float x)

double **g_acos** (double x)

float **g_acosf** (float x)

double **g_asin** (double x)

float **g_asinf** (float x)

double **g_atan** (double x)

float **g_atanf** (float x)

double **g_atan2** (double y, double x)

float **g_atan2f** (float y, float x)

double **g_floor** (double x)

float **g_floorf** (float x)

double **g_fmod** (double x, double y)

float **g_fmodf** (float x, float y)

double **g_ceil** (double x)

float **g_ceilf** (float x)

double **g_exp** (double x)

float **g_expf** (float x)

double **g_log** (double x)

float **g_logf** (float x)

double **g_log10** (double x)

float **g_log10f** (float x)

Additionally to these basic math functions some other functions are provided:

void **g_check_bounds** (double *t, double a, double b)
corrects `t` to the nearest point within the interval [`a`, `b`]

int **g_circle** (VEC3 m, VEC3 u, VEC3 p, VEC3 q, double t, VEC3 *res)
calculates a point on the circular arc around `m` from `p` to `q` where `u` defines the plane's normal, the resulting point is returned in `res`

int **g_gcd** (int m, int n)
gives the greatest common divisor for the integers `m` and `n`

int **g_helix** (VEC3 a, VEC3 b, VEC3 p, VEC3 q, double t, VEC3 *res)
calculates a point on the helix from `p` to `q` with axes `a` and `b`, the resulting point is returned in `res`

10.2.16 Functions for Triang2d

For the `TRIANG2D` structure see 6.3.1.

`int` **g_get_opp_vertex** `(INT3 vertex, int node1, int node2)`
: return index of the third point in the triangle (i.e. return `vertex[i]` if `vertex[i]` != `node1/node2`) or -1 if the function fails

`int` **g_get_opp_vertex_ind** `(INT3 vert, int node1, int node2)`
: return local index of the third point in the triangle (i.e. return `i` if `vertex[i]` != `node1/node2`) or -1 on failure

`int` **g_get_opposite_vertex** `(TRIANG2D *tr2d, int triangle_ind, int node_ind)`
: return the index of the point in the neighbour triangle (opposite to `node_ind`) which has `triangle_ind` as neighbour

`int` **g_get_triangle_with_edge** `(TRIANG2D *tr2d, int node1, int node2)`
: return the number of a triangle within tr2d that contains both vertices `node1` and `node2`

`int` **g_triang2d_get_cls_pnt** `(TRIANG2D *tr2d, VEC3 p)`
: return the index of a point within `tr2d` as close as possible to `p`

`int` **g_triang2d_get_cls_not** `(TRIANG2D *tr2d, VEC3 p, int ind)`
: return the index of a point within `tr2d` as close as possible to `p` which is not `tr2d ->x/y/z[ind]`

`double` **g_triang2d_get_edge_length** `(TRIANG2D *tr2d, int tr_ind, int node_ind)`
: return the length of the edge that is opposite to the `node_ind`'th vertex of the `tr_ind`'th triangle within `tr2d`

`double` **g_triang2d_get_size** `(TRIANG2D *tr2d)`
: compute the maximal distance between points of the triangulation

`double` **g_triang2d_get_vertex_angle** `(TRIANG2D *tr2d, int tr_ind, int node_ind)`
: return angle point with index `node_ind` in the triangle `tr_ind`

`int` **g_triang2d_point_equal** `(TRIANG2D *tr2d, int p_ind, int q_ind, double eps)`
: return 1 if the distance between the points with indices `p_ind` and `q_ind` is smaller than `eps`

`double` **g_triang2d_point_sqrdist** `(TRIANG2D *tr2d, int p_ind, int q_ind)`
: returns the square of the distance between the points with indices `p_ind` and `q_ind`.

10.2.17 Complex Functions

In the following we list a number of complex functions available in **GRAPE** together with some useful complex constants.

10.2.17.1 Complex Constants

`g_CZERO` = {`0., 0.`}

`g_CE` = {`1., 0.`}

`g_CI` = {`0., 1.`}

`g_CNE` = {`-1., 0.`}

`g_CNI` = {`0., -1.`}

`g_CPI4` = {`0.70710678, 0.70710678`}

g_C3PI4 = {-0.70710678, 0.70710678}

g_C5PI4 = {-0.70710678, 0.70710678}

g_C7PI4 = {0.70710678, -0.70710678}

10.2.17.2 Complex Functions with one Complex Argument

COMPLEX **g_cneg** (COMPLEX u)
 -u

COMPLEX **g_cinv** (COMPLEX u)
 1 / u

COMPLEX **g_cconj** (COMPLEX u)
 conjugate of u

double **g_creal** (COMPLEX u)
 Re(u)

double **g_cimag** (COMPLEX u)
 Im(u)

double **g_cnorm** (COMPLEX u)
 |u|

double **g_carg** (COMPLEX u)
 argument of u, return value is -pi < carg(u) <= +pi

COMPLEX **g_cexp** (COMPLEX u)
 exp(u)

COMPLEX **g_csin** (COMPLEX u)
 sin(u)

COMPLEX **g_ccos** (COMPLEX u)
 cos(u)

COMPLEX **g_ctan** (COMPLEX u)
 tan(u)

COMPLEX **g_ccot** (COMPLEX u)
 cot(u)

COMPLEX **g_csinh** (COMPLEX u)
 sinh(u)

COMPLEX **g_ccosh** (COMPLEX u)
 cosh(u)

COMPLEX **g_ctanh** (COMPLEX u)
 tanh(u)

COMPLEX **g_ccoth** (COMPLEX u)
 coth(u)

COMPLEX **g_clog** (COMPLEX u)
 logarithm of u, undefined on the negative x-axis

COMPLEX **g_csquare** (COMPLEX u)
 u * u

`COMPLEX` **g_ccube** `(COMPLEX u)`
`u * u * u`

`COMPLEX` **g_csqrt** `(COMPLEX u)`
square root of `u`, undefined on the negative x-axis

10.2.17.3 Complex Functions with two Complex Arguments

`COMPLEX` **g_cadd** `(COMPLEX u, COMPLEX v)`
`u + v`

`COMPLEX` **g_csub** `(COMPLEX u, COMPLEX v)`
`u - v`

`COMPLEX` **g_cmul** `(COMPLEX u, COMPLEX v)`
`u * v`

`COMPLEX` **g_cdiv** `(COMPLEX u, COMPLEX v)`
`u / v`

`COMPLEX` **g_cpow** `(COMPLEX u, COMPLEX w)`
`u` to the power of `w`

10.2.17.4 Complex Functions with one Complex and one Real Argument

`COMPLEX` **g_cradd** `(COMPLEX u, double r)`
`u + r`

`COMPLEX` **g_crsub** `(COMPLEX u, double r)`
`u - r`

`COMPLEX` **g_crmul** `(COMPLEX u, double r)`
`r * u`

`COMPLEX` **g_crdiv** `(COMPLEX u, double r)`
`u / r`

`COMPLEX` **g_crpow** `(COMPLEX u, double r)`
`u` to the power of `r`

`COMPLEX` **g_casqrt** `(COMPLEX u, double r)`
square root of `u`, undefined on the line with angle `r`

`COMPLEX` **g_crot** `(COMPLEX u, double phi)`
rotation of `u` by the angle `phi`

10.2.17.5 Other Complex Functions

`COMPLEX` **g_cmplx** `(double x, double y)`
make a complex number with real part `x` and imaginary part `y`

`COMPLEX` **g_cpolar** `(double r, double phi)`
transformation from polar coordinates into complex numbers

`COMPLEX` **g_cmoebius** `(COMPLEX a, COMPLEX b, COMPLEX c, COMPLEX d, COMPLEX u)`
moebius transformation of the complex number `u`, `a - d` all complex

`COMPLEX` **g_crmoebius** `(double a, double b, double c, double d, COMPLEX u)`
moebius transformation of the complex number `u`, `a - d` all real

`COMPLEX` **g_cpower** `(COMPLEX u, int n)`
`u` to the power of `n` with integer `n`

`COMPLEX` **g_croot** `(COMPLEX u, int n)`
`n`th root of `u`, undefined on the negative x-axis

`COMPLEX` **g_cSKP** `(COMPLEX u[], COMPLEX v[], int n)`
`n`-dimensional complex scalar product

`COMPLEX` **g_cBIL** `(COMPLEX u[], COMPLEX v[], int n)`
multiplication of two `n`-dimensional complex vectors

`COMPLEX` **g_cSKP3** `(CVEC3 u, CVEC3 v)`
3-dimensional complex scalar product

`COMPLEX` **g_cBIL3** `(CVEC3 u, CVEC3 v)`
multiplication of two 3-dimensional complex vectors

Bibliography

[AVS] C. Upson et al: *The Application Visualization System: A Computational Environment for Scientific Visualization,* IEEE Computer Graphics and Appl., July 1989

[Bra] K. Brakke: The surface evolver, *Exp. Math. 2,* 141-165 (1992).

[Geo] M. Phillips et al: *The Geomview Manual*, The Geometry Center, Minneapolis, 1995

[Kar1] H. Karcher: The triply periodic minimal surfaces of Alan Schoen and their constant mean curvature companions, *Manuscripta Math. 64* (1989), 291–357

[Kar2] H. Karcher: Construction of Minimal Surfaces, *Surveys in Geometry*, Tokyo, 1989

[New1] *GRAPE Newsletter 1*, Sfb 256 Universität Bonn, December 1993

[New2] *GRAPE Newsletter 2*, Sfb 256 Universität Bonn, June 1994

[OP] B. Oberknapp, K. Polthier: A Simple Concept for Distributed Computing in Computer Graphics, *Modeling - Virtual Worlds - Distributed Graphics*, Infix-Verlag, Sankt Augustin, 1995

[PP] U. Pinkall, K. Polthier: Computing discrete minimal surfaces and their conjugates, *Experimental Mathematics,* Vol. 2 (1993), No. 1

[XDR] eXternal Data Representation, *Sun Technical Notes (for RPC 4.0),* Mountain View, California, Sun Microsystems, Inc.

Index

GRAPE Classes and Types

GRAPE Methods Sorted by Classes

GRAPE Methods Sorted by Name