GRAPE

GRAphics Programming Environment

Manual

Sonderforschungsbereich 256
Institut für Angewandte Mathematik
Universität Bonn

Institut für Angewandte Mathematik
Universität Freiburg

Version 5.0

Copyright © by the Sonderforschungsbereich 256 at the Institut für Angewandte Mathematik, Universität Bonn.

Permission is granted to make verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Contents

Preface **7**

1 Introduction **9**
1.1 What is GRAPE? . . . 9
1.2 Difference to Existing Programs . . . 10

2 What Basic Concepts are Implemented **11**
2.1 Triangulations And Refinement Methods . . . 11
2.2 Displaying Data Hierarchies . . . 12
2.3 Time Dependent Structures . . . 12
2.4 Extending Grape . . . 13
2.5 Projects . . . 13

3 How to use a Grape Program **15**
3.1 Using the Demo Programs . . . 15
3.2 Different Types of Interactive Devices . . . 15
3.2.1 The Mouse in GRAPE . . . 15
3.2.2 Layer . . . 16
3.2.3 Button . . . 16
3.2.4 Inputs . . . 17
3.2.5 Slider . . . 18
3.2.6 Ruler . . . 18
3.2.7 Time-slider and Time-ruler . . . 18
3.2.8 Scrollbar . . . 19
3.2.9 List . . . 20
3.2.10 Spline-Editor . . . 20
3.3 Manager and Graphic Device . . . 21
3.3.1 Graphic Device . . . 21
3.3.2 Control Device . . . 21
3.4 Controlling and Viewing . . . 21
3.4.1 The Five GRAPE-Menus . . . 22
3.4.2 General Interaction Strategy . . . 22
3.4.3 Get the Feeling . . . 23
3.4.4 The “Manager” Menu . . . 27
3.4.5 The “Light” Menu . . . 32
3.4.6 The “Surface” Menu . . . 32
3.4.7 The “Trans” Menu . . . 35
3.4.8 The “Options” Menus . . . 39
3.4.9 Output Facilities . . . 39
3.5 The Concept of Time . . . 43
3.5.1 What Does “time” Mean? . . . 43
3.5.2 Classes Managing Dynamic Processes . . . 44

3.5.3 Creating Classes That Implement Dynamic Processes . . . 44
3.5.4 The Class TimeStep Extending Key-Frame Techniques . . . 45
3.5.5 Adaptive Strategy for Timedependent PDE . . . 45
3.5.6 Time Dependent Interactives . . . 46
3.6 Projects . . . 47
3.6.1 What is a Project? . . . 47
3.6.2 Advantages of the Concept . . . 47
3.6.3 Using a Project . . . 47
3.6.4 Overview of Existing Projects . . . 49

4 Programming with Grape **51**
4.1 Object-Oriented Programming . . . 51
4.1.1 Sequential Versus Object-Oriented Programming . . . 51
4.1.2 Classes, Instances and Methods . . . 51
4.1.3 Main Concepts of Object-Oriented Programming . . . 52
4.2 GRAPE . . . 52
4.2.1 The `GRAPE` Function . . . 52
4.2.2 Memory Management in GRAPE . . . 54
4.2.3 GRAPE Datatypes and Utility Functions . . . 54
4.2.4 GRAPE Math Functions . . . 54
4.3 Creating a New Class . . . 55
4.3.1 Design and Implementation . . . 55
4.3.2 Naming Conventions . . . 55
4.3.3 The Data Structure for a New Class . . . 55
4.3.4 Adding a Class to the System . . . 56
4.3.5 The `Rot2d` Example . . . 57
4.4 Writing Methods . . . 58
4.4.1 Class and Instance Methods . . . 58
4.4.2 Adding a Method to the System . . . 59
4.4.3 Exception Handling . . . 60
4.4.4 Memory Management . . . 60
4.4.5 Copying Instances . . . 64
4.4.6 Display Methods . . . 67
4.4.7 Methods for Debugging . . . 69
4.4.8 The `"universal"` Method . . . 71
4.4.9 More Methods for Rot2d . . . 72
4.5 Interactive Elements . . . 77
4.5.1 What is Available . . . 77
4.5.2 Creating Items . . . 78
4.5.3 The `new_item` Function . . . 78
4.5.4 Adding Items to the Manager . . . 83
4.5.5 Display Methods Using Items . . . 84
4.5.6 A User-Interface for Rot2d . . . 85
4.5.7 Putting It All Together . . . 87
4.5.8 Problems of the Interface . . . 89
4.6 Projects . . . 89
4.6.1 How Projects Work . . . 90
4.6.2 Creating Projects . . . 90
4.6.3 The `"setup"` Method . . . 91
4.6.4 Using or Adding a Project . . . 93
4.6.5 “Unusing” or Removing a Project . . . 94
4.6.6 Project Rotate . . . 95
4.6.7 A Standard Main Program . . . 102
4.6.8 Changes From the Previous Version . . . 103

5 Geometry Classes **105**
5.1 Points . . . 105
5.2 Curves . . . 105
5.2.1 Triang1d . . . 105
5.2.2 Bnd-Curve . . . 109
5.2.3 Cons . . . 111
5.3 Surfaces and 2D Finite Elements . . . 114
5.3.1 Triang2d . . . 114
5.3.2 Fe2d . . . 124
5.3.3 Adapt2d . . . 127
5.3.4 Geom2d . . . 130
5.4 Volumes and 3D Finite Elements . . . 134
5.4.1 Triang3d . . . 135
5.4.2 Fe3d . . . 140
5.5 Moving and Tracing . . . 144
5.5.1 The Concept of Moving and Tracing . . . 144
5.5.2 Calculating a Move . . . 146
5.5.3 Visualizing the Trace . . . 149
5.5.4 Clouds . . . 150
5.5.5 Streaklines . . . 151
5.6 General Meshes . . . 153
5.6.1 Revisiting numerical data and visualization requirements . . . 153
5.6.2 Description of the mesh concept . . . 154
5.6.3 Methods on Mesh3d . . . 159
5.6.4 Conversion Methods . . . 161
5.6.5 Examples for the implementation of element types . . . 163
5.6.6 Mesh2D . . . 167
5.6.7 Methods on Mesh2d . . . 169
5.7 Using the colorbar . . . 169
5.7.1 The structure of colorbar . . . 170
5.7.2 Methods on colorbar . . . 171

6 Supervising Classes **173**
6.1 The Data Hierarchy Tree . . . 173
6.1.1 Scenes — A Collection of Geometry Objects . . . 173
6.1.2 Chain . . . 176
6.1.3 TimeScene . . . 179
6.2 TimeStep . . . 180
6.2.1 General methods on TimeStep . . . 181
6.3 Dynamic Objects Supervising TimeStep . . . 183
6.3.1 Time-Object . . . 183
6.3.2 Surface . . . 188

7 Projects **195**
7.1 Projects for Continuum-Mechanics . . . 195
7.1.1 Project Move-Trace . . . 195
7.1.2 The Grid Editing Projects . . . 196
7.1.3 Project Probe . . . 204
7.2 Projects for Differential Geometry . . . 205
7.2.1 Project Time-Object . . . 205
7.2.2 Project Curve . . . 208
7.2.3 Project Surface . . . 211
7.2.4 Project Surface-Build . . . 213
7.2.5 Project Explicit . . . 218

7.2.6 Project Amandus . . . 222
7.2.7 Project Dual . . . 226
7.2.8 Project Diri . . . 227
7.3 Misc Projects . . . 227
7.3.1 Project Model . . . 227
7.3.2 RGBVideo . . . 227
7.3.3 Video production and GRAPE . . . 230
7.3.4 Scene Tree . . . 232

8 Interactive Classes and Devices 235
8.1 Interactive Classes . . . 235
8.1.1 General Structure . . . 235
8.1.2 Class Hierarchy . . . 237
8.1.3 Classes and Methods . . . 237
8.1.4 Utility Functions . . . 266
8.2 Graphic Devices . . . 267
8.2.1 Device . . . 267
8.2.2 GraphicDevice . . . 267
8.2.3 An Example Low–Level Display Method . . . 274
8.2.4 GraphicVideo . . . 277
8.2.5 ControlDevice . . . 278
8.3 Text Devices . . . 281
8.3.1 Text-Devices . . . 281
8.3.2 Primitive Text Functions . . . 282

9 Utility Classes and Functions 285
9.1 Utility Classes . . . 285
9.1.1 Linked Lists . . . 285
9.1.2 List-Of-Inst . . . 288
9.2 Utility Functions . . . 290
9.2.1 Memory Functions . . . 290
9.2.2 Functions on VEC3 . . . 292
9.2.3 Functions on DVEC3 . . . 293
9.2.4 Functions on DVEC4 . . . 294
9.2.5 Matrix Functions . . . 296
9.2.6 Functions for Arrays and n-dim Vectors . . . 297
9.2.7 Transformation Functions . . . 298
9.2.8 String Functions . . . 298
9.2.9 Swap Functions . . . 299
9.2.10 Drawing Routines . . . 299
9.2.11 Math Utility Functions . . . 299
9.2.12 Functions for Triang2d . . . 300
9.2.13 Complex Functions . . . 301

Index 305

List of Figures

3.1 Manager Menu 28
3.2 Light Menu 33
3.3 Surface Menu 34
3.4 Trans Menu 36
3.5 Camera 38
3.6 Option Menu 40
3.7 PostScript Layer and Its Gamma Curve Editor 41
3.8 Paper Size Layer 42
3.9 Dynamic Instance of a TimeScene 45

5.1 example 144
5.2 the different scales 146
5.3 moving process 147
5.4 tracing 150

7.1 Project Dependencies 195
7.2 User Interface for Project Time_Object 205
7.3 Display Layer for Project Surface 211
7.4 User Interface for Project Surface 212
7.5 Reflection Config-Layer in Project Surface 213
7.6 User Interface of Project Dual 226
7.7 User Interface of Project RGBVideo, Containing the Images Part 228
7.8 Main User Interface of Project Scene Tree 232

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. To keep up with these developments, we decided to publish the GRAPE Manual in a fifth — totally revised — edition.

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

Bonn, Freiburg · October, 1995

The GRAPE–Team

E–Mail: `grape@iam.uni-bonn.de`
WWW: `http://www.iam.uni-bonn.de/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, time-dependent vector fields on surfaces and in volumes, etc. Non-mathematical classes exist for graphics drivers and the viewer together with control elements such as buttons, rulers and spline editors for example.

The mathematical algorithms consist for example of tools for computing parametric curves and surfaces, level set surfaces, minimal and H-surfaces generated using different algorithms, evolution of surfaces and of curves restricted to spatial surfaces, solution of ordinary and partial differential equations, operations on vector fields on surfaces and in volumes, operations on geometric objects in Euclidean, spherical and hyperbolic space, evaluation of finite element functions on geometric objects and several other operations.

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.

Besides standard viewing facilities, visualization tools include e.g. various clipping algorithms, extraction of level lines and level surfaces, interactive refine and coarsen operations, reflection tools, calculating particle traces, display of functions on surfaces or volumes, vector fields, picking, editing grids. Most routines work cooperatively with time-dependent geometries.

Time-dependency is an advanced new concept in 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. Many algorithms are also 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/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

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 4.1.

2.1 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 5.3 and 5.4.

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

2.2 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

2.3 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

2.4 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 4 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

2.5 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 3.6.4 for a short overview, chapter 7 for a detailed description. Using projects is easy, you just have to know how to handle them with the GRAPE manager (see section 3.6.3). For programmers using a project has many advantages, an example can be found in chapter 4.6.

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 3

How to use a Grape Program

Compiling and using a Grape program will be explained by using a demo program which allows to explore all the basic features Grape offers.

3.1 Using the Demo Programs

To create the demo executable you need the demo sources and the corresponding makefile from the GRAPE distribution. Ask the person who has installed GRAPE — probably the system administrator — where the libraries and the demo directories can be found. Perhaps there is already a demo executable installed, which can be useful to learn how to use the GRAPE menus, but this won't help you to write a makefile for your own GRAPE applications.

Copy the makefile and the demo source you want to compile from the demo directory to a working directory. Follow the instructions in the makefile to adapt it to your system and your GRAPE installation. You need to know where the GRAPE libraries and header files can be found and what kind of system you are working on. We can't explain in this manual how makefiles are used to create an executable. If you don't know how to use a makefile to compile a C-program ask someone who is familiar with it.

If you have successfully created the demo program (or you were lucky to find one) just start it. If everything works correctly, two new windows will appear on your display. If this does not work you should check the display variable and the paths for the GRAPE library and the header files. If they are ok and you still can't get the program started you should try to find someone who is already working with GRAPE and might be able to help.

3.2 Different Types of Interactive Devices

Before we start to describe the GRAPE-Control 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 an GRAPE-Program. After that the different interactives and all there usage will be described.

3.2.1 The Mouse in GRAPE

GRAPE makes use of three-mouse (if your system doesn't have so much the will be emulated will special-keys). The left mouse-button is the "normal" button of GRAPE and will be used in the for the normal action. The middle mouse-button is the cancel-button and it could be used to cancel the actual action immediately. The right mouse button is used for an alternative use or to open a configuration-menu for the interactive. (This will be described at the ... interactives.)

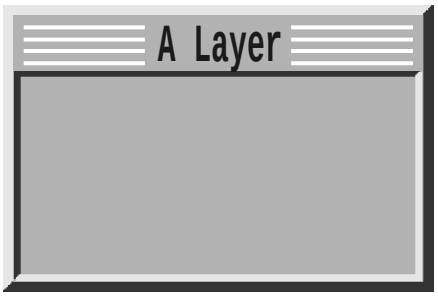
A GRAPE Program distinguishes between to different mouse-actions "click" and "drag". With "click we mean that you pressed down the mouse button and the release it without moving the

mouse. It is not necessary to be really at the same place, but it is necessary that you are inside of the same interactive when releasing the mouse. Otherwise the action you started will be canceled. With "dragging" we mean, that you press down a mouse button and then move the mouse to a new position, while keeping the button pressed. The action will end, if you release the button at all. If you drag it is not necessary to be inside the object at the moment of releasing.

You can always control only one object at the same time. This means that you have to release all mouse-buttons first before you can chose an other object. As long as GRAPE is working or you keep a mouse button pressed, the mouse cursor will have the look of a clock. An exception are the interactives which wait for key-presses. (We think you won't be very happy, if you must keep the mouse-buttons pressed while typing a text....)

3.2.2 Layer

Layers in GRAPE To get a more useful and logical menu-structure several parts of the control-element may appear in a layer. A layer will pop up and lay above the normal GRAPE-menus. An empty layer (without any control-elements) will look like that:



A layer behaves like a window inside a normal window. You can move it by dragging it's border with the left mouse button. Otherwise you can resize it by dragging it's 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 dragging it with the right one. A shrinked layer will look like like the headline of the layer. The other parts of the layer will be hidden. Most layers have got a button named "done", "dismiss" or "close" to remove the layer.

3.2.3 Button

buttons are the most used control-elements in GRAPE. They are used for switching between the menus, starting action and several other things. A normal buttons looks like the that:

You can select it by simply click on it with the left mouse button. (Remember that your action is cancelled if you release the button while the mouse is outside the button or just press the middle mouse button.) While you keep hold of the mouse button the button will be pressed and what you see is:

Be aware of buttons, which look like the following one, because these buttons are disabled and can't be pressed.

There are (up to now) two special types of buttons: Cycle-button: This button will change it label each time you click on it. So with that button you can select one of several options be

clicking on the button as much as you need to select the option like like. Up to now it is not possible to see if a button is a cycle-button or not. Checkboxes: With a checkbox you can switch things on or off. They are just simple switches you can use. If a checkbox can be switched off, then it will look like that one:

Alternative if it looks like the following one it is switched on:

3.2.4 Inputs

An Input is a field where you can enter text or numbers. If an input is not selected it looks like a button. The different could be seen only if you click on it. After that an input will look like this:

Now you've got two possibilities to act. The first one is to start typing the new content of the input, so the old one will be destroyed after you've pressed the first key:

The other possibility is to use one of the control-keys (cursor left/right) first. Doing you can edit the actual content of the input.

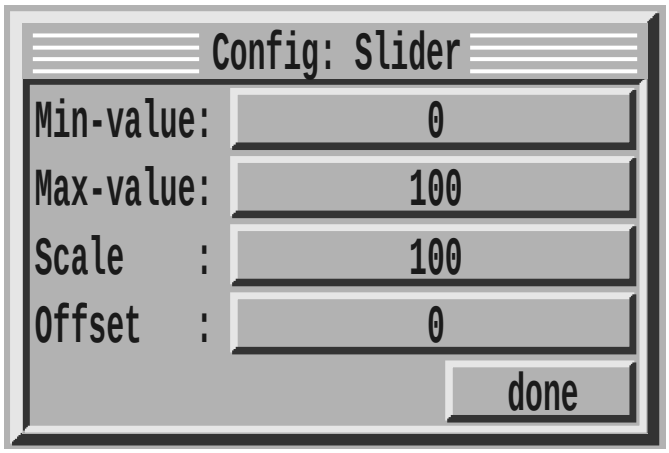
You end your typing by "enter" or click the input with the left mouse button again. If you press the middle mouse button or select an other input the old content of the actual input will be restored. The following control-keys can be used (up to now):

- cursor left/right moves cursor one character backward/forward
- home/end moves cursor to start or end of line
- delete deletes character at cursor position
- backspace deletes character backward
- escape clears the input
- enter end of typing
- tab the input will make a word-completion

3.2.5 Slider

A slider is used to control fixed values between a minimal an a maximal value, often used for percent values. A simple slider will look like this:

You've got several possibilities to set the slider to a new value. First of all you can select the input at right side of the slider and type in the new value directly. Secondly you can click or drag the bar at the left side of the slider with the left mouse button to a new position. (Remember that you can can restore the old slider-value if you press the middle mouse button before releasing the left one.) The input will always show the actual value of the slider. If you click the slider with the right mouse button the following config-layer will pop-up:



In this config-layer you can configure the slider like you prefer it. The Min-/Max-Values are the minimal and maximal values you can set with the slider. If you try to enter a value or to drag the slider to a value out of this range it will be automatically corrected. The scale is the range the slider shows from his left to his right side. Finally the offset is the value at the left side of the slider. So the value at the right side of the slider is offset+scale.

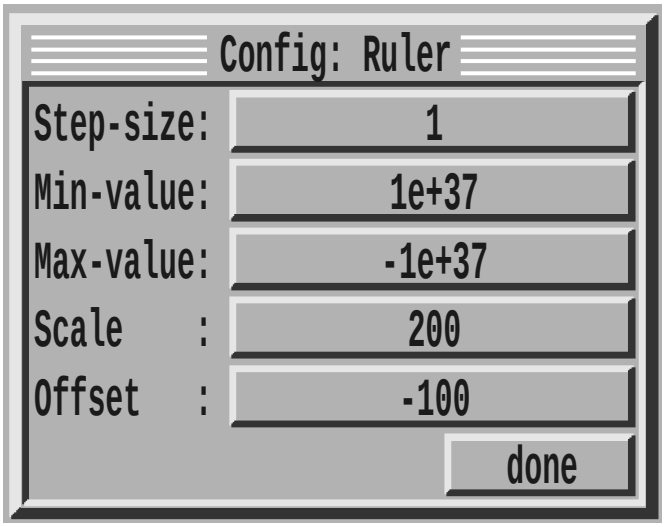
3.2.6 Ruler

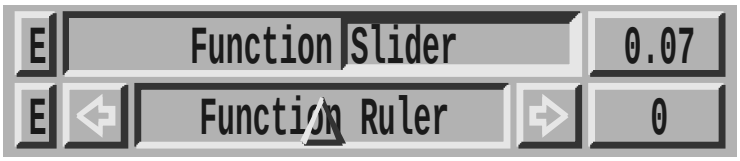
A ruler is used to control various values. A simple ruler will look like this:

Additional to the slider a ruler has got two step-button, which are used to set the value to multiples of a ruler defined step-size. You can set the value of a ruler in the same way as the value of a slider (by click, drag or input). The configuration-layer of a ruler contains one more input, in which you can change the set-size.

3.2.7 Time-slider and Time-ruler

There are also special kinds of slider and ruler which can be time-depended and are called Function slider/ruler. You can distinguish these sliders and rulers from normal ones by their additional Button named "E" at the left side of the ruler:

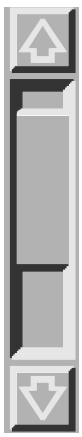




With this button you can pop-up a spline-editor for changing the time-depend function of the ruler and switch the dependence on and off. A detailed description of the spline-editor will follow further below. If the dependence is on this button will be drawn pressed down and the slider/ruler will change its value by evaluation its spline after time has been changed.

3.2.8 Scrollbar

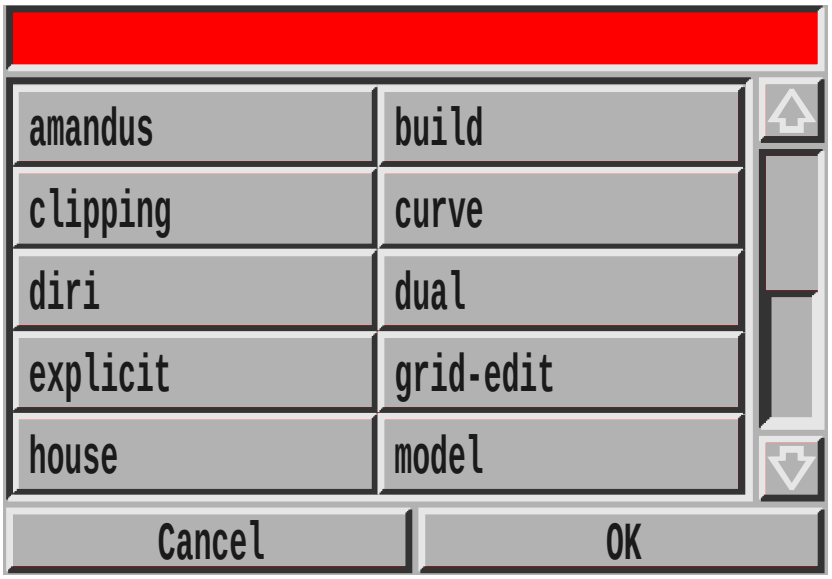
As the last control element similar to ruler we will describe the use of a scroll-bar. A typical scroll-bar will look like this:



A scrollbar normally is used to scroll a list of filenames or methods. The size of the bar shows you how much of the total number of entries are visible. By scroll the bar up and down you can select which part of the list you will see. You can scroll the bar by dragging it or scroll linewise with the step-buttons. A Click above or beneath the bar will scroll pagewise.

3.2.9 List

A list is designed to select on of several possibilities, like files or methods. A list will look like this:

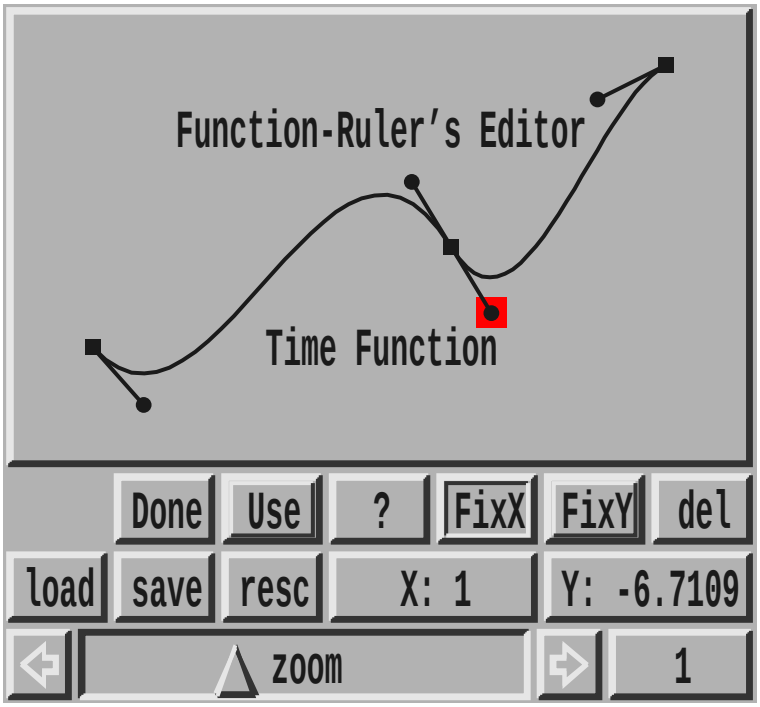


The top-line of the list is made up off an input in which you can select the item you what by typing its name. Too the input will show the item you've actual selected. At the right border the list owns a scrollbar which is used to scroll the list if you can't see all items at ones. To select an item of the list you must click it with the left mouse button and then click the button named OK. Alternative you can click the item with the left mouse button twice, this is the same as click it and then click the button named OK. If you click with the right mouse button in the list of items the list will scroll up or down. The direction of scrolling depends on your position inside the list of item. The button named cancel in the left corner is used to cancel the list, so that for most lists the list will disappear

3.2.10 Spline-Editor

The spline-editor is used to define the time-dependence function of time-dependent sliders and ruler.

You can change the spline by simply dragging its points or the tangent of the point to the new value you like. An other possibility is to select a point with a click and enter its new coordinates in the inputs named x and y. To enter new points in the spline you simply have to click at the chosen position with the right mouse button. Remove them by keep the shift key pressed and click them with the right mouse button. After that short description about changing the spline we will give a brief description of the buttons, inputs and the ruler at the bottom of the editor. Done - close the spline-Editor Use - Switches time dependence of the ruler on or off ? - show a brief help for the mouse button usage in the spline-editor FixX/ FixY - If one of these checkboxes is pressed you can't edit the coordinate of the spline-points. 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: - Input which show the coordinates of the actual chosen point zoom - Zooming for the Editor (reset with rescale button)



3.3 Manager and Graphic Device

GRAPE has got two machine dependent devices. With this concept its is possible to use GRAPE on a great number of different systems. Also this concept will make use of different hardware possibilities like 3d-graphicboards.

3.3.1 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 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 8.2.2.

3.3.2 Control Device

The control device is used from the manager to show his menus and get his user events. In the actual GRAPE version only a X11 control device is supported, because all up to now supported machines do support X11 too. The manger controls also your graphic device. With the manager you can change the preferences of the actual graphic device like grip or patch mode, doublebuffer and several other things. The manager could never be reviewed without the graphic device, because a lot of action you make in the manager will 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 will be described in the next section.

3.4 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 colours, 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

3.4.1 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.

The menus are designed to fulfill different tasks:

`light`	Create and manipulate lightsources illuminating the scenery. Their colours, directions, and positions can be influenced.
`surf`	Edit surface properties. Surface properties define colouring and shading of surfaces.
`trans`	Move, scale or rotate the scenery; this affects the current transformation matrix.
`mngr`	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).

3.4.2 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.

3.4.3 Get the Feeling

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

You have created your first executable in section 3. 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 th 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 the cycle button until its label is 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.
	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.
"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 graphis window because the TimeScene has an own Scene matrix that is unused now.
"Tree" TimeScene	Go back onto the TimeScene.
scene time	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.
scene time (I)	Initialize the time dependency spline.
scene time (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.
scene time (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 Splie_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 3.6.

The next sections describe everything in detail.

3.4.4 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.

3.4.4.1 General Functions

`Project: add`	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 7.
`Project: del`	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 3.4.9.1). 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.

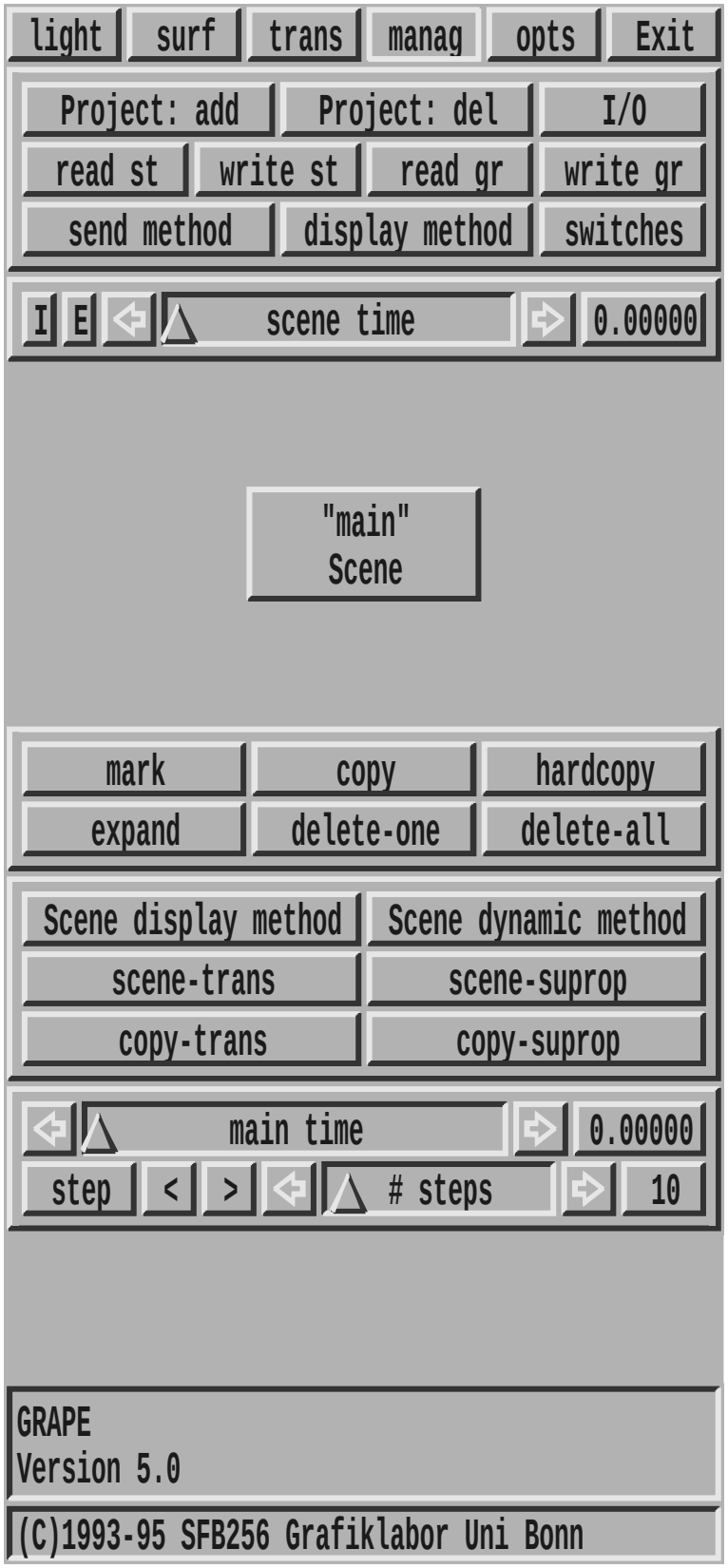


Figure 3.1: Manager Menu

`write st` Write the current graphic-STate as a file `*.st`. It consists of the current state of the light, surf, and trans menu with exception of the camera. 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 geometrical data in GRAPE file format.

`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.

`disp 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. (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.
- `Double-Buffer` 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.
- `Depth-Cue` 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.
- `view-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.

3.4.4.2 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.

`scene time` functionruler — Using this ruler you can set the local time for the last visited TimeScene. 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.

`main time` ruler — Using this ruler you can walk to a time between 0 and 1.

`step` — Selection button to choose the behaviour of the direction buttons `<` and `>`. 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*. The middle mouse button is used to stop the motion. When a bound of the time interval is passed, the animation starts over at the other one, thus looping through the animation.

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.

`<`, `>` — Walk through the time in discrete steps. The total number of steps is given by the following ruler:

`#steps` ruler — Controls the number of subdivisions of the time interval $[0, 1]$.

3.4.4.3 Hierarchy Management

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).

Building up a scene tree interactively is enabled by the following buttons:

`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.

`del obj` 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.

`del 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.

3.4.4.4 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 6.1.1) 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 6.1.3). 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` Scroll through the three matrix modes projection, viewing and modelling. "no matrix" means the matrix is unused.
- `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 (3.4.6).

- `emission`,`ambient`,`diffuse`,`specular` 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.
- `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.

load, save	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.

3.4.5 The "Light" Menu

An important aspect of visualization is the use of colour. 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 do see holes is to put a position lightsource inside.

3.4.5.1 The lightsources

0, ..., 8	These buttons represent the different lightsources. 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.
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: 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.
light-color	Compose a colour for the active lightsource with this color editor. (about color editors see Surface menu (3.4.6)).
	Edit the position/direction with the same Interactives as in the trans-menu (3.4.7). Note that the position and distance rulers disappear when editing direction lightsources.
light intensity factor	Functionslider for intensity scale factor for all lightsources. Useful for fading effects in animated scenes.

3.4.6 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.

Figure 3.2: Light Menu

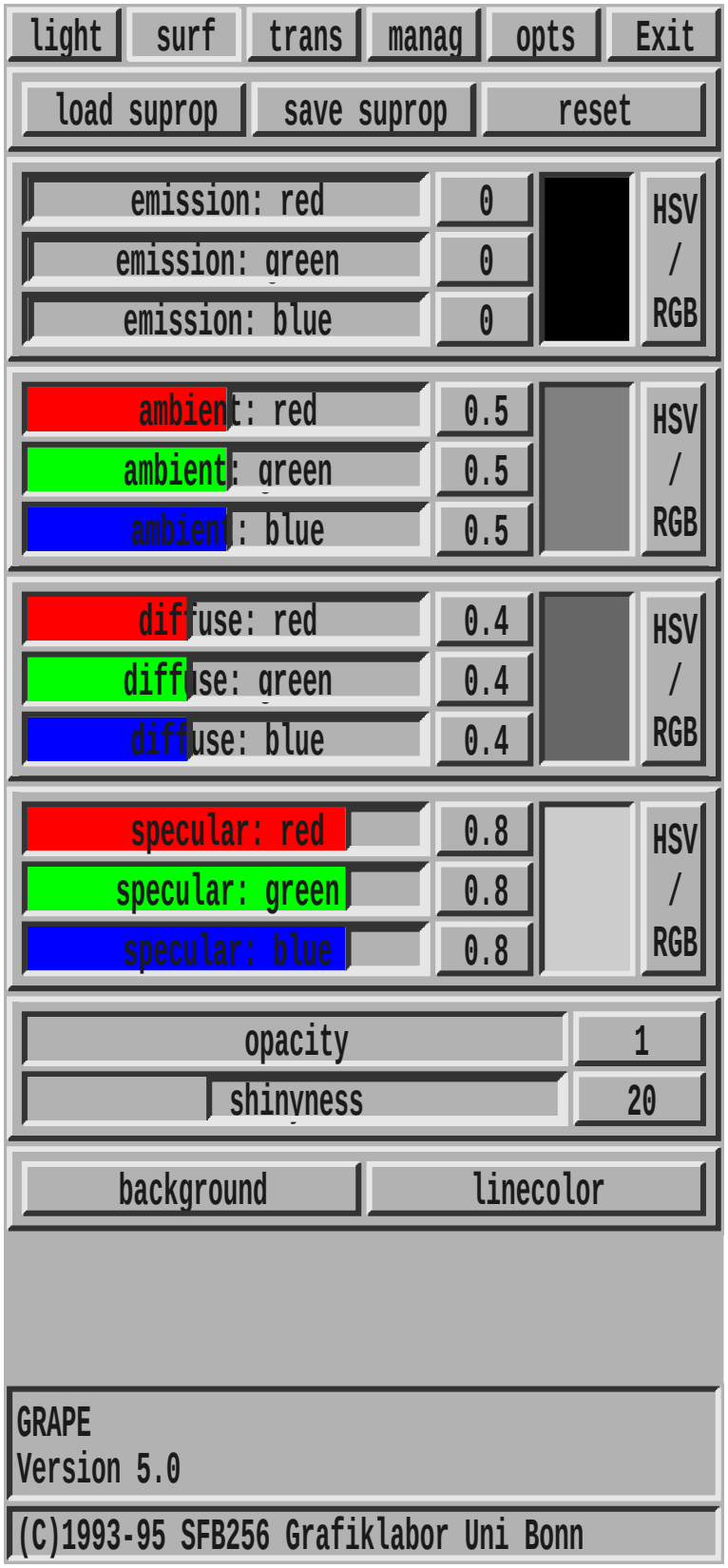


Figure 3.3: Surface Menu

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

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.
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:
shinyness	Values for this parameter range from 0 to 59. It describes the inverse size of the highlights.
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.
load suprop, save suprop	these buttons open file selectors which give you the uniue opportunity to load and save SUrface PROPerties.
background, linecolor	these buttons open additional color editors for the global background and linecolor settings respectively. Press dismiss to close them.
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.

3.4.7 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-Transform, camera-trans	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.

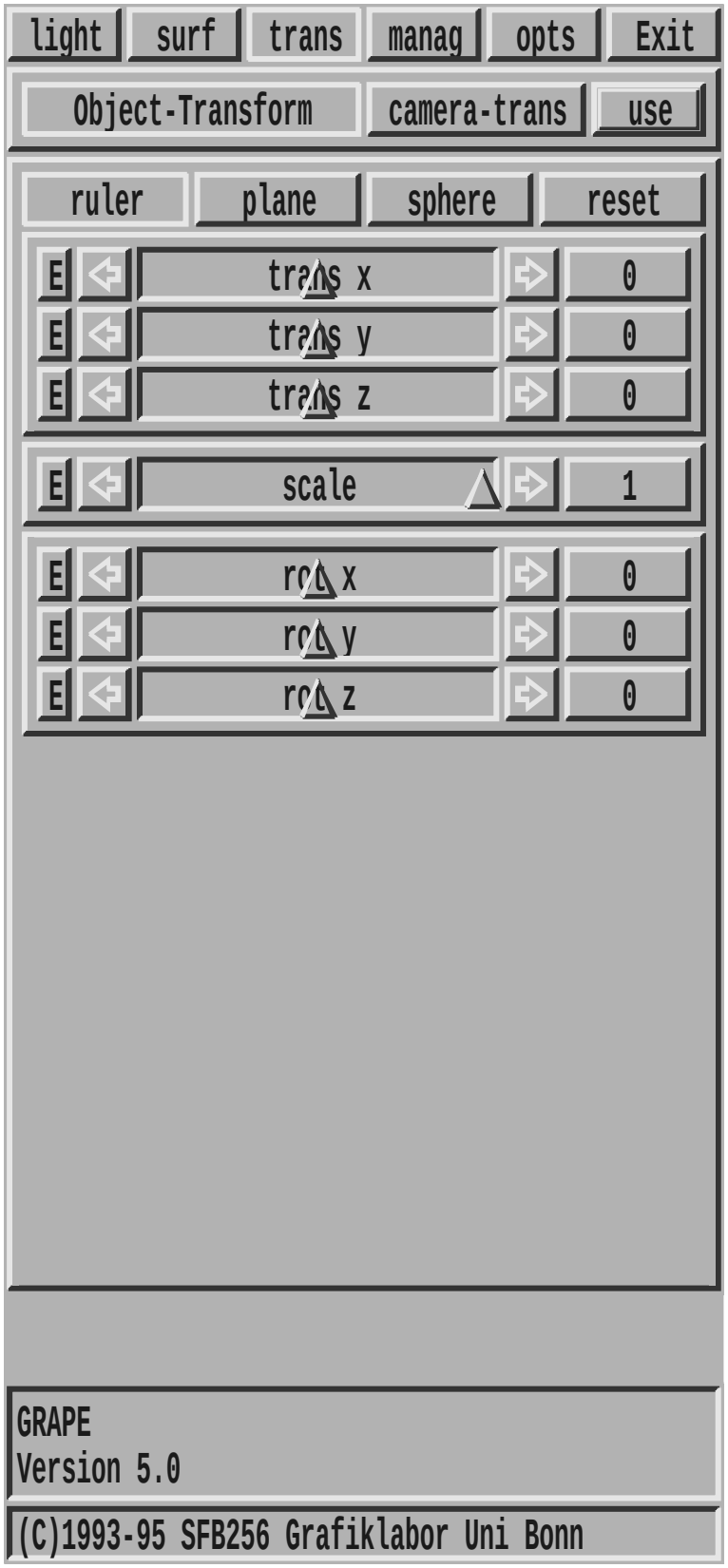


Figure 3.4: Trans Menu

3.4.7.1 Object Transformation

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

3.4.7.2 Translation

`trans x`

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

`scale` Scales the size of the object.

3.4.7.3 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.

3.4.7.4 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 (3.4.7).

`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.

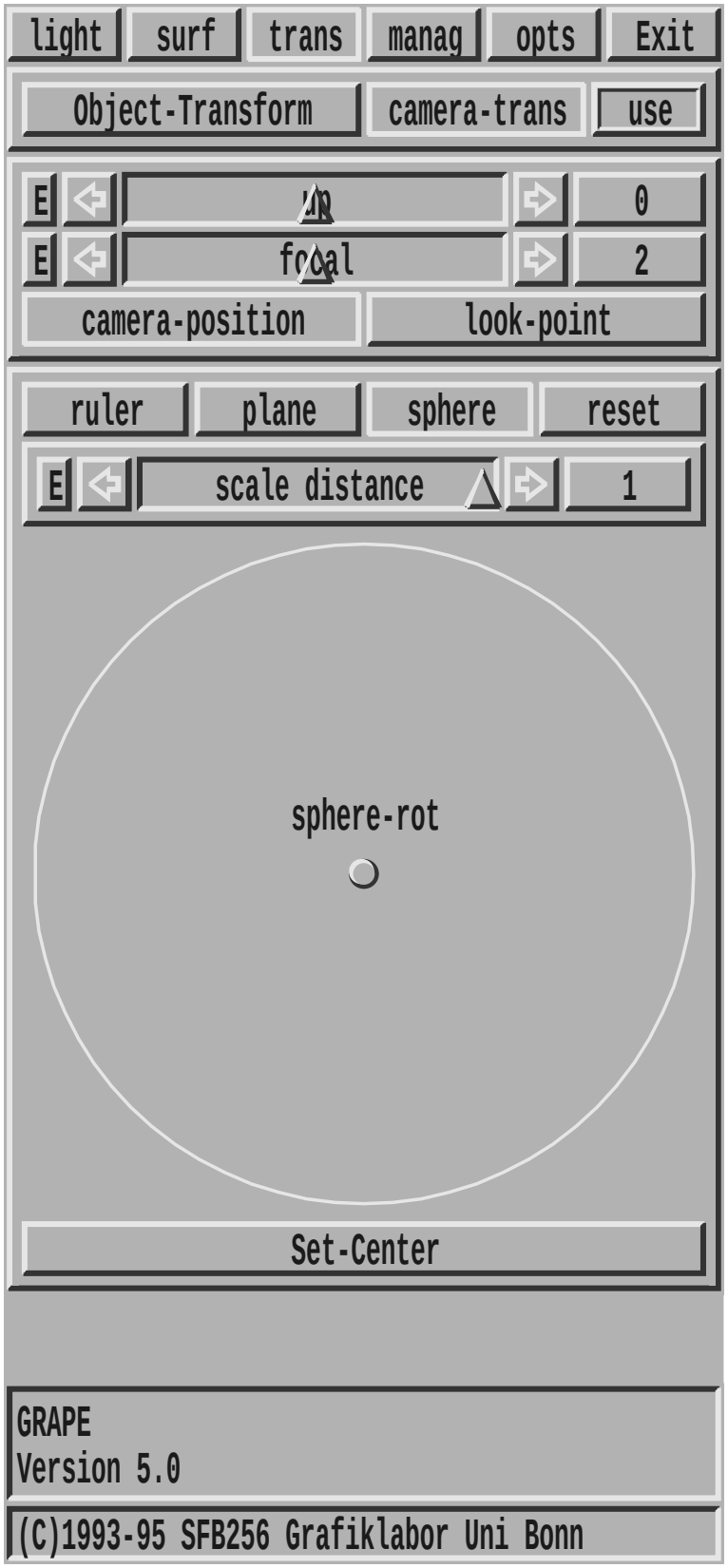


Figure 3.5: Camera

`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.

3.4.8 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 7.

3.4.9 Output Facilities

3.4.9.1 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.

These settings can be read from a file, though not very comfortable yet. GRAPE searches a file `./ps.opt`, then `~/.ps.opt`, the later described values are read as single numbers, refer to the shell output when first using the driver after start (some obsolete features are supported here and some names are changed in this layer). The default contents of this file are:

```
0 1 1
843 596 72
0 1 1
0.3 0.59 0.11
0
1.0 0 1
0 0
```

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.

Even more advanced users can try to speed up by omitting intersection check or to stop a running long time needing intersection checking process by creating a file `zbuff.raw` or `zbuff.stop` in the current directory.

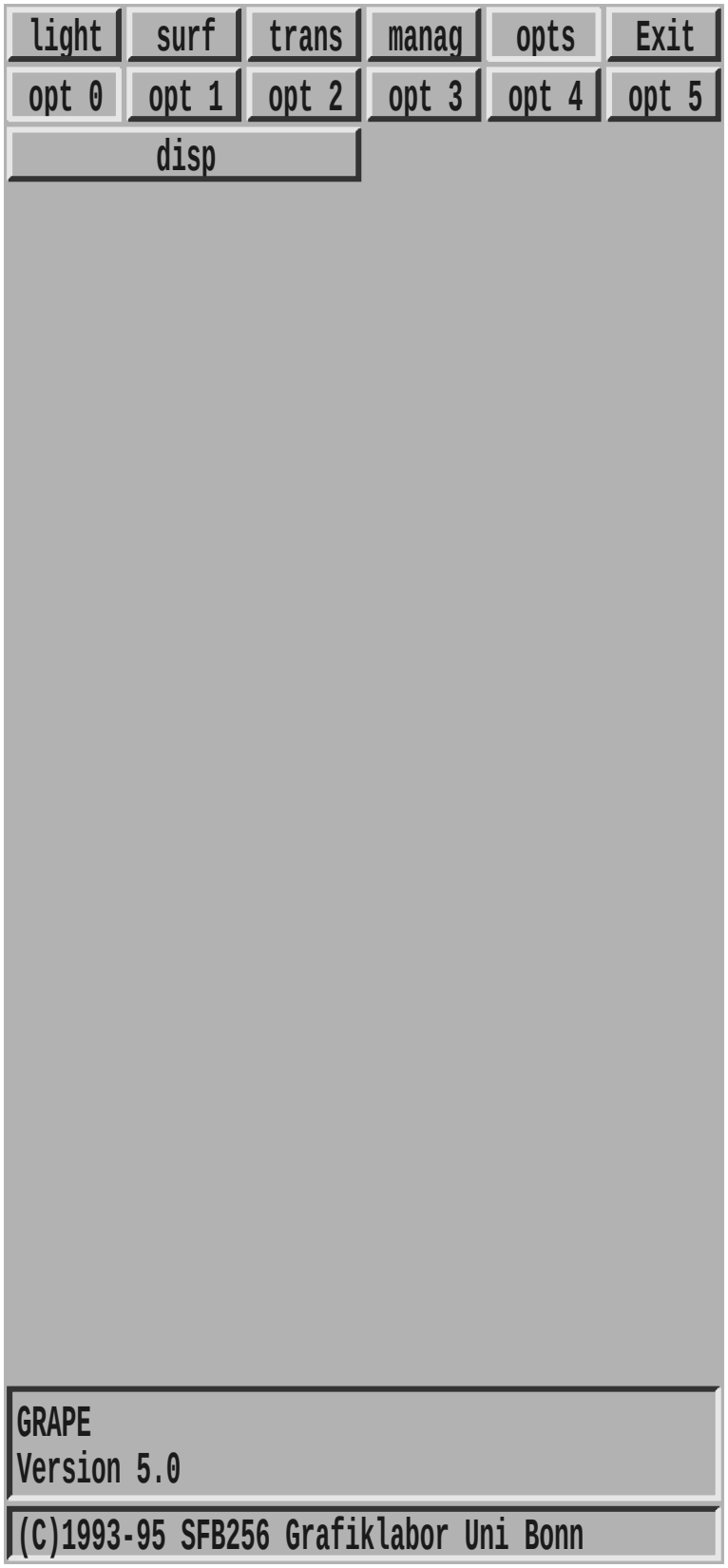


Figure 3.6: Option Menu

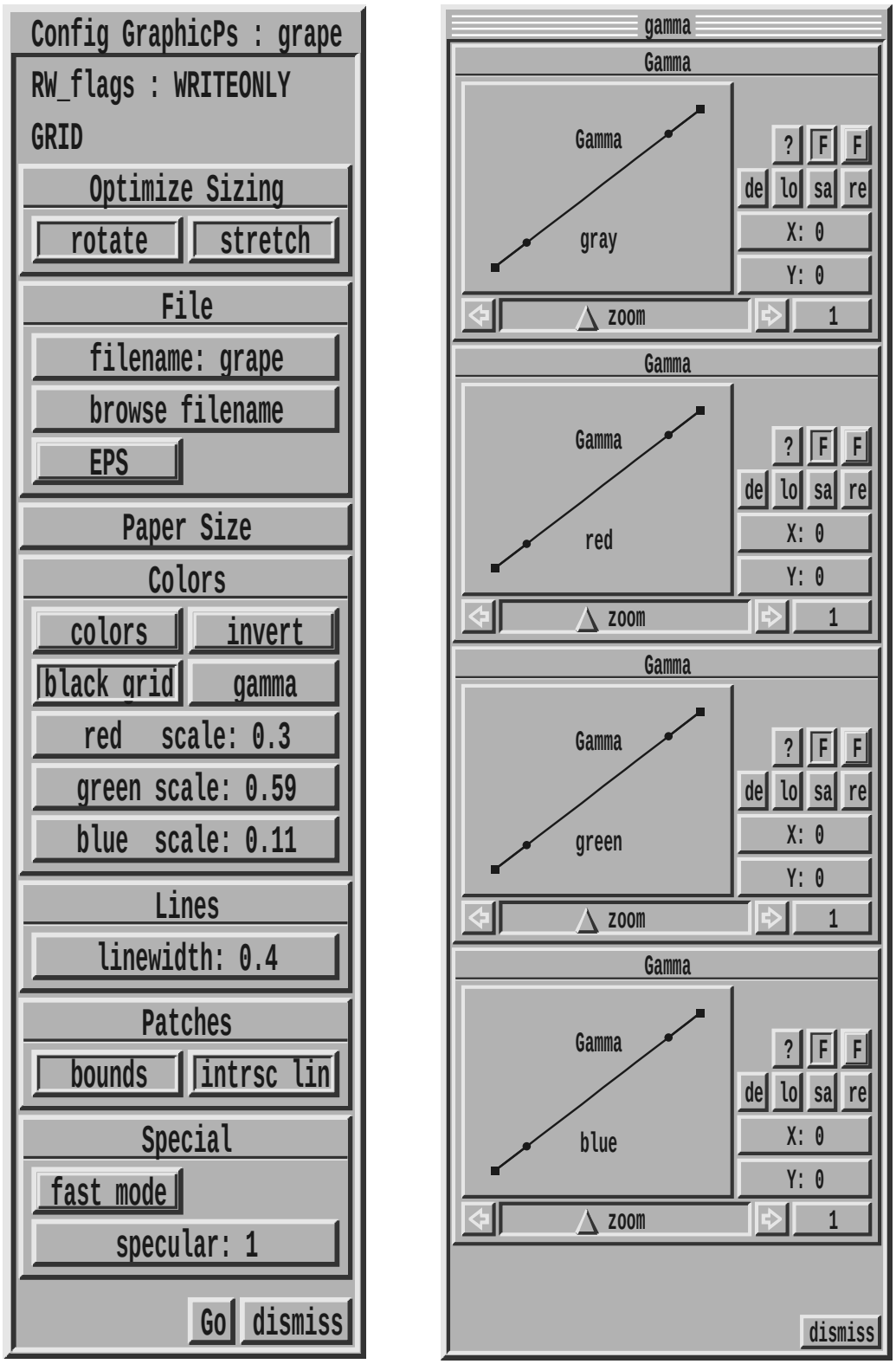


Figure 3.7: PostScript Layer and Its Gamma Curve Editor

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. 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 3.8: 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 sub-layers. 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.

3.4.9.2 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 7.

3.5 The Concept of Time

3.5.1 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.

3.5.2 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.

3.5.2.1 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.

3.5.2.2 Time_Object

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

3.5.2.3 Surface

Surface is a subclass of Time_Scene. 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.

3.5.3 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 Time_Object and Surface require a certain type of data structure as dynamic process, therefore they can not be used in this generality.

3.5.4 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.

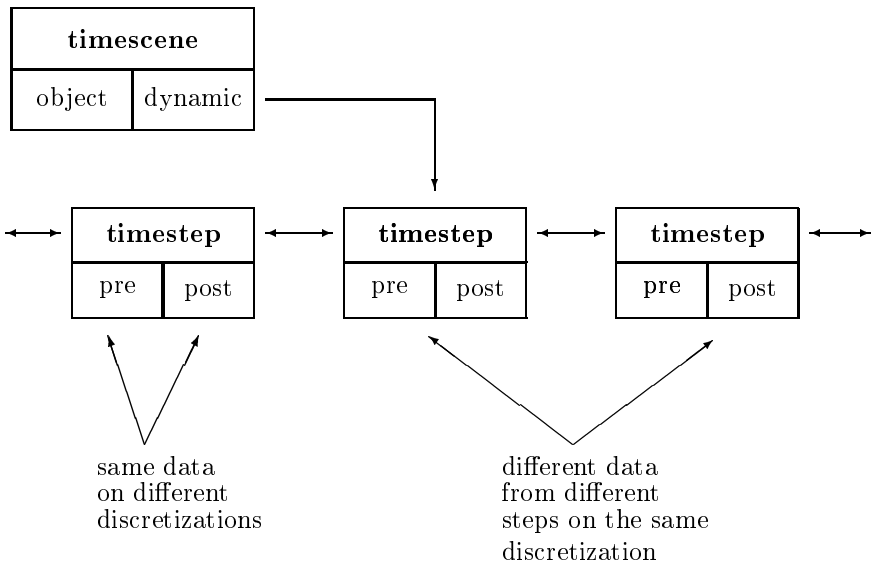


Figure 3.9: Dynamic Instance of a TimeScene

3.5.5 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

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

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 3.9.

3.5.6 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.

3.6 Projects

One of the fundamental concepts in GRAPE is it 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 7.

More information about the internal project mechanisms and details about writing projects can be found in section 4.6 in the chapter "Programming with GRAPE". There have been some changes to the project management since the previous GRAPE version, therefore if you have written your own project using the old project management, you should carefully read section 4.6.8.

3.6.1 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 usefull, 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 menues even if the user is working on something else and they are not needed. A seperate 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 get more complicated with each application that is added, and if packages are exchanged between several users it get unmanageable.

The solution for these problems are projects. A projects 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, they will automatically be loaded resp. unloaded with the project. New objects can be inserted into the data hierarchy when a project is added.

3.6.2 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 things a user has to do to add a project to his executable is to link the files of the project (or even better: a library with these files) to her/his 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 4.6.

3.6.3 Using a Project

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

When the add buttons 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 something like this:

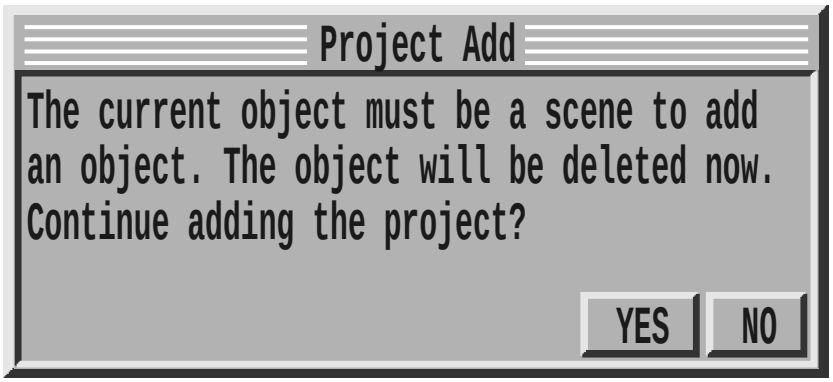
Now you can select a project to add it to the system (the same project can be added several times though this only makes sense for projects that add objects into the data hierarchy). Removing projects is as simple as adding them: press the delete button and a list of all added projects appears, then 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 (this is inconvenient and will change in the next release).

Projects can be declared as hidden, then they don't 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 implements an abstract superclass, it can only be used together with one of its subclasses).

What happens when the project is added depends on the specific project. The first thing you might notice is that new objects appear in the data hierarchy. Projects that add objects need a scene node to be able to do this, therefore if you add such a project while the current object is no scene an error requester will appear

and you can select to continue adding the project (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 menues, where you can find it and how the project interface is used is described in the project's documentation in section 7. There you will also find a documentation of the classes and methods that are available with the project.



3.6.4 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 full documentation in section 7 is given. There you will find a description of the projects user-interface and its classes and methods.

Amandus (7.2.6): Compute parameter-dependent minimal surfaces from their Weierstrass representation. The parameters and domain information is read from a description file and can be edited interactively.

Build (7.2.4): 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 (7.2.2): Compute parameter-dependent curves from expclicitly given formulas read in from a description file.

Diri (7.2.8): Combines *Build*, *Refine* and *Dual* to an environment for computing discrete minimal surfaces and their conjugates.

Dual (7.2.7): The Dirichlet minimizer for computing discrete minimal surfaces and their conjugates. Starting triangulations can be created for example with *Build*.

Explicit (7.2.5): Compute parameter-dependent surfaces given by explicit formulas. The formulas, parameters and information about the domain is read from a description file and can be edited interactively.

Grid_Edit (7.1.2.2):

House (no documentation): Demo project which inserts a geometry into the data hierarchy.

Images (7.3.2.1): Internal project used by *RGBVideo*, it implements the file handling.

Model (7.3.1): "Hard" transformation of geometries. In contrast to the surface menu or the scene transformations the object's coordinates are transformed.

Move_Trace (7.1.1):

Probe (7.1.3):

Refine (no documentation): Interactive curvature and degeneracy dependent refinement. Included are methods to refine the boundary or the area inside a ball.

RGBVideo (7.3.2): Project for creating bitmap files of arbitrary size and video sequences. Only available on SGI machines at the moment.

Scene_Tree (7.3.4): Data hierarchy viewer, very useful when working with large hierarchies.

Surface (7.2.3): Basic project for working with parameter-dependent surfaces. Several geometries can be handled at the same time, the parameters are read from a description file.

Tensor_Inspect (no documentation):

Time_Object (7.2.1): Basic project for working with parameter-dependent geometries. The parameter are read from a description file.

Topo (no documentation):

Tr2d_Edit (7.1.2.1):

Chapter 4

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.

4.1 Object-Oriented Programming

4.1.1 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.

4.1.2 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.

4.1.3 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".

4.2 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.

4.2.1 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, ...)
```

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 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 3.4.2), 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 4.4.6). 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 4.4.6 and 4.5.5).

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 4.4.8), 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 4.4. 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 → double, short int → int, char → 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.

4.2.2 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.

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 9.2.1. When implementing a new datatype you should also provide the corresponding memory functions.

4.2.3 GRAPE Datatypes and Utility Functions

Additionally to some utility classes like `G_List` for arbitrary linked lists (see 9.1.1) and classes for arrays of objects like `List_Of_Inst` (see 9.1.2) 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`), float (`VEC2`, `VEC3`, `VEC4`) and double (`DVEC2`, `DVEC3`, `DVEC4`) variables, three by three and four by four matrices (`MATRIX33` and `MATRIX44` for float, `DMATRIX44` 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 9.2 to see what's available.

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.

4.2.4 GRAPE Math Functions

In section 9.2.11 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.

4.3 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.

4.3.1 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 methods 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.

4.3.2 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.

4.3.3 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`):

```
typedef struct {
  /* SYSTEM ONLY */
  CLASS *class;      <- instance variables of class Root
  char  *name;
  INSTANCE *next;
  unsigned int refcount;
  /* SUPERCLASS  */
```

```
  ...                <- instance variables of the superclass
  /* NEW_CLASS */
  ...                <- instance variables of the new class
} NEW_CLASS;
```

This definition 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.

All class type definitions must contain the `/* SYSTEM ONLY */` part, these are the instance variables of the base class `Root` of GRAPE's class hierarchy. These variables are for internal use only, except for some special cases described below they must be accessed by methods only.

4.3.4 Adding a Class to the System

Together with the methods described in section 4.4.2 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(class, "new-class")(name, size)

CLASS *class
char *name
int size

This will create a subclass of `class` 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.

Classes can be deleted by calling

GRAPE(class, "delete")()

CLASS *class

The class `class` will be removed from the class hierarchy only if there are neither subclasses nor instances of `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 = (CLASS *)GRAPE(obj, "delete")();
```

Unfortunately there still exist `"delete"` methods which return the pointer to the instance or class (this pointer is invalid by now!) to indicate that the method was successful, therefore it might be a good idea to reset the pointer by hand if it will be used again later:

```
GRAPE(obj, "delete")();
obj = NULL;
```

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.

4.3.5 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`.

Now this is the structure for out new class. Up to the comment `/* ROT2D */` it is a copy of the `TRIANG2D` structure. 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):

```
typedef struct {
  /* INSTANCE */
  CLASS *class;
  char *name;
  INSTANCE *next;
  unsigned int refcount;
  /* TRIANG2D */
  int number_of_points;
  int max_number_of_points;
  float *x, *y, *z;
  int number_of_elements;
  int max_number_of_elements;
  INT3 *vertex;
  INT3 *neighbour;
  /* 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 */
} 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 4.5.7):

```
#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 4.4.3.

4.4 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.

4.4.1 Class and Instance Methods

Methods can be send to classes as well as to instances. Both types of methods basically have the same format:

```
TYPE *class_method_name(parm1, parm2, ...)
{
  TYPE *self;

  self = (TYPE *)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 *`.

`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 == (INSTANCE *)self->class) {
  /* called on class */
  ...
} else {
  /* called on instance */
  ...
}
```

`self` should be declared as `INSTANCE *` in this case.

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 4.2.1.

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 4.4.3 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"`. 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.

4.4.2 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

(char *)GRAPE(class, "find-method")(name)

(char *)GRAPE(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.

4.4.3 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 the general error handling macro

```
ASSURE(<expression>, "error message", <statement>)
```

which continues execution only if the `<expression>` is `TRUE`. Otherwise the `"error message"` string is printed together with the corresponding file name and line number and the user is prompted for a response. Then the `<statement>` is executed.

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

```
END_METHOD(NULL)
```

for `<statement>`. Because in C a block bounded by `{ }` is treated as a single statement it is possible to put a whole sequence of commands into the macro. This can for example be used 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);
       });
```

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

4.4.4 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.

4.4.4.1 `"new-instance"` and `"delete"`

These methods 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 4.5.2 and 8.1). 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 has to free all memory that was allocated for its instance variables (not only the memory allocated by `"new-instance"`). By calling `"delete"` on the superclass the superclass' instance variables are freed and finally `Root "new-instance"` will free the memory for the instance struct.

The memory of the instance can only be freed by `"delete"` if the instance's reference counter is zero, that is the instance referenced from nowhere else. If there is more that one reference only the `refcount` is decremented (see example below and section 4.4.5.2).

In both cases `"delete"` should return `NULL`, then the return value of `"delete"` can be used to reset the pointer to the instance that was deleted (compare `"delete"` on classes 4.3.4).

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_delete()
{
  ROT2D *self;

  /*
   * This is an instance method, therefore G_INSTANCE is used.
   * Again the return value is checked, somebody might have called
   * "delete" on the class Rot2d (or a subclass).
   */
  self = (ROT2D *)START_METHOD(G_INSTANCE);
  ASSURE(self, "", END_METHOD(NULL));

  /*
   * Two cases have to be distinguished in a delete method depending
   * on the value of the reference counter of the instance:
   */
  if(self->refcount)
    /*
     * If the reference counter in not zero the instance is used
     * somewere else and can not be deleted. We decrement the ref-
     * counter instead to mark that this reference to the instance
     * isn't used any more.
     */
    self->refcount--;
  else {
    /*
     * The instance can now be deleted, but we have to delete curve
```

```
     * and axis first.
     */
    if(self->curve)
      GRAPE(self->curve, "delete")();
    if(self->axis)
      GRAPE(self->axis, "delete")();

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

  /*
   * Delete 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 4.5.7):

```
#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")("delete", rot2d_delete);

  ...
}
```

4.4.4.2 `"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.

Unfortunately also the functionality of the `"list-alloc"` methods differs from class to class. For `Triang1d`, `Triang2d` and `Triang3d` old data is always preserved: if there is already enough memory nothing happens, otherwise new lists are allocated and the old data is copied before the old lists are freed. For other classes (even for some subclasses) this is still not the case, then the old data will be lost.

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.

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.

4.4.5 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 3.4.4.3).

4.4.5.1 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 3.4.4.3).

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 to default curves 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);
  }
```

4.4.5.2 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 (if `refcount` is zero).

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.

4.4.6 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 3.4.2): 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 4.5.5.

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 float 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]), dfFLOAT,
                                   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 8.2.2.

4.4.7 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.

4.4.7.1 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`.

4.4.7.2 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.

4.4.7.3 The Methods `"get-info"` and `"follow"`

If you have worked with the GRAPE manager before or read section 3.4.4.3 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);
}
```

4.4.8 The "universal" Method

As described in section 4.2.1 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.

4.4.9 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 first have to free the
   * old memory lists, then we allocate longer arrays.
   */
  GRAPE(self, "list-free")();
  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.

```
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: this might be different for "list-alloc" on other
   * classes.
   */
  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.
   */
  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.

4.5 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.

4.5.1 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 8.1):

- `Button`s (8.1.3.2) 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 (8.1.3.4) 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` (8.1.3.3) — a collection of `Checkbox`es — can be used.

- `Input`s (8.1.3.12) 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 (8.1.3.20) 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 (8.1.3.9) are time-depended rulers. Their variable is controlled by a spline which can be edited with the `Spline_Editor` (8.1.3.24) that is associated to the function-ruler.

- `FileList`s (8.1.3.7) 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` (8.1.3.22) 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_Se`lectors (8.1.3.5), which allow editing a color given by RGB values in RGB or HSV mode, or `Matrix_Edit`ors (8.1.3.17) which control a transformation matrix. The basic classes for these composite elements are `Group` (8.1.3.11) and `Layer` (8.1.3.15).

Most of these interactives were already described in the introduction in section 3.2, the full documentation of all interactive classes and their methods can be found in chapter 8.1.

4.5.2 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 4.5.8 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 8.1.

4.5.3 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 4.6.6.1.

4.5.3.1 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 8.1.

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_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_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 3.2.6).

`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_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_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_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 3.2.2).

`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`, see 4.6.8), 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`.

4.5.4 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_FillMode`) 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 8.1.

4.5.5 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 4.4.6: 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.

4.5.6 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", dfFLOAT);
ypos = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->curve->y[self->current], "y pos", dfFLOAT);
zpos = (RULER *)GRAPE(Ruler, "new-instance")
  (&self->curve->z[self->current], "z pos", dfFLOAT);
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.

4.5.7 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")("delete", rot2d_delete);
  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;
}
```

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.

4.5.8 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.

4.6 Projects

What projects are and how they are used interactively was already explained in section 3.6. 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.

4.6.1 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).

4.6.2 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 4.6.7 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 `"delete"` method will display an error message when you accidentally try to delete a project.

4.6.3 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 4.6.6.2.

`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 4.3.4), 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_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 4.6.6.2). Since every method has its own class (pointer) pointer methods from different classes can be handled with one list.

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 4.5.3), 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, see section 4.6.8.

`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`.

4.6.4 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. In batch mode the manager isn't available and it wouldn't make sense to add the items. 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 4.6.6.4 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 4.6.6.3).

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.

4.6.5 "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*.

4.6.6 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.

4.6.6.1 A New User-Interface

As explained in section 4.5.8 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 4.5.3) 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 float 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, dfFLOAT,
     I_RSizeX, 1.0, I_End);
    ypos = (RULER *)new_item(Ruler, I_Name, "y pos", I_Var, &ytmp, dfFLOAT,
     I_RSizeX, 1.0, I_End);
    zpos = (RULER *)new_item(Ruler, I_Name, "z pos", I_Var, &ztmp, dfFLOAT,
     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.

4.6.6.2 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, "delete",             (FUNC)rot2d_delete            },
  { &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, "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, 1, 0,
       P_Date, "Sep 20, 95",
       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.

4.6.6.3 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);
}
```

4.6.6.4 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.

4.6.7 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 4.5.

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.

4.6.8 Changes From the Previous Version

The project management was completely rewritten, it now provides much better functionality. Some small changes therefore were unavoidable.

Project `"new-instance"` now has a new parameter list, therefore these calls have to be edited. The project setup-method must not be called by hand any more, calling it might still work but it could get you into trouble. The project's setup-method is now only called when the project is used for the first time, but that should make no difference.

The new `g_project_include` function should now be used for adding the projects, then `"new-instance"` and the project setup-method don't have to be called directly and (almost) nothing can go wrong.

You have to edit the add-method of the project, because the object to be added to the data hierarchy has to be set by `"set-object"` now, the `scene` instance variable is no longer available.

The old `"set"` method is still there, but its parameter list has changed (the slider pointer was added):

```
GRAPE(project, "set")(PROJECT_DESCR *projects, CLASS_DESCR *classes,
                      METHOD_DESCR *methods, RULER_DESCR *rulers,
                      BUTTON_DESCR *buttons, SLIDER_DESCR *sliders,
                      char *init_method, char *exit_method, char *add_method,
                      SCENE *scene, char *path);
```

Converting it to the new `"setup"` method is easy: just replace `"set"` by `"setup"`, remove all `NULL` pointers and insert the tags described in section 4.6.3 into the list (for example `P_MethodList` for `methods`, `P_AddMethod` for `add_method` and so on).

The scene parameter is no longer needed, the project will insert an object if `"set-object"` is called. The path wasn't used, it has been removed.

Chapter 5

Geometry Classes

5.1 Points

We have no special class Point. Since this would cause too much overhead we suggest the usage of the structure VEC2 or DVEC2 in two space, VEC3 or DVEC3 in three space and VEC4 or DVEC4 in four space when working with single points. Be aware that both structures are no **GRAPE** classes, they are defined as

```
typedef VEC2 float[2];
typedef DVEC2 double[2];
typedef VEC3 float[3];
typedef DVEC3 double[3];
typedef VEC4 float[4];
typedef DVEC4 double[4];
```

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.

5.2 Curves

The simplest classes for geometries in **GRAPE** are for curves. The mainly used class for curves is Triang1d, whose entries consist of a sequence of points in three space. For this class a number of display methods exist to use its data e.g. as ordinary lines, or with a tube around, or as arrows.

The sub class Bnd_Curve is used for curves on surfaces when the points of the curve have a direct connection with the surface, e.g. a boundary curve of a surface patch.

Further curve representations are the class Bezier1d and Spline1d. Currently they are only used by the FunctionRuler (8.1.3.9) and the SplineEditor.

Additionally to these static data structures exist time dependent curves of class Curve described in chapter Projects.

5.2.1 Triang1d

The class Triang1d contains purely data information:

```
typedef struct triang1d {
    /* SYSTEM ONLY */
    CLASS *class;
    char *name;
```

```
    INSTANCE *next;
    unsigned int refcount;
    /* TRIANG1D */
    int number_of_points;     /* currently used number of points */
    int max_number_of_points; /* length of list vectors */
    float *x, *y, *z;         /* point coordinates */
} TRIANG1D;
```

5.2.1.1 Memory Methods on Triang1d

GRAPE(triang1d, "delete")()

TRIANG1D *triang1d;

The associated memory of the instance variables is freed–up and the instance is deleted. This only takes place when triang1d ->refcount is zero, otherwise the reference counter is decreased by one. For a more detailed description of this event see the description of the "delete" method on other classes.

GRAPE(triang1d, "list-alloc")(max-points)

GRAPE(triang1d, "list-free")()

TRIANG1D *triang1d;
int max-points;

"list-alloc" allocates memory for max-points points for the coordinate lists, "list-free" frees the memory for the coordinate lists.

GRAPE(triang1d, "list-shrink-send")()

TRIANG1D *triang1d;

Free unused space of the vertex lists: new vertex lists for triang1d ->number_of_points points are allocated, the points with index from 0 to triang1d ->number_of_points are copied and the old vertex lists are freed. This method is called by

GRAPE(triang1d, "optimize-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). "hardcopy" may not be sent to superclasses since always a new structure is allocated. The pointer to the new Triang1d 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 Triang1d 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. If the method fails it returns NULL.

5.2.1.2 Display Methods on Triang1d

GRAPE(triang, "display")()

TRIANG1D *tr

The instance tr is rendered on the standard device in a default manner.

GRAPE(triang, "pipe-disp")()

GRAPE(triang, "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 */
float scal ; /* factor for scaling */
int diskr ; /* parameter for discretization */
float 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")()

GRAPE(triang1d, "tube")(top, tail, loc-diskr, loc-scal)

```
TRIANG1D *triang1d;
int top, tail; /* no meaning up to now */
int loc-diskr; /* discretization */
float loc-scal; /* radius */
```

The instance triang1d is rendered on the standard device in a default manner. "tube-disp" calls "tube" (which is a faster and less flexible version of "pipe") to draw a pipe 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".

5.2.1.3 I/O Methods on Triang1d

GRAPE(triang1d, "write")()

TRIANG1D *triang1d;

The Triang1d instance is written to the standard textdevice. The output format is described in chapter ->Grape-Files. A NULL pointer is returned if an error occurs.

(TRIANG1D *)GRAPE(Triang1d, "read")()

CLASS *Triang1d;

A Triang1d instance is read from the standard textdevice. A NULL instance is given back if an error occurs.

5.2.1.4 Other Methods on Triang1d

GRAPE(triang1d, "interpol")(t1_one, t1_two, factor)

GRAPE(triang1d, "interpol-fac")(t1_one, t1_two, fac1, fac2)

TRIANG1D *triang1d ;
TRIANG1D *t1_one, t1_two;
double factor, fac1, fac2;

Linear interpolation of the two Triang1d instances t1_one and t1_two with factor:

triang1d = (1 - factor) * t1_one + factor * t1_two,

or with factors fac1 and fac2:

triang1d = fac1 * t1_one + fac2 * t1_two.

If necessary new vertex lists are allocated for triang1d.

The two Triang1d instances are interpolated pointwise, they should have the same number_of_points.

GRAPE(triang1d, "reflect")(trfmat, orient)

TRIANG1D *triang1d;
MATRIX44 trfmat;
int orient;

Create a new Triang1d instance from triang1d with triang1d ->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(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 has the outlook of a C source file.

GRAPE(triang1d, "reset")()

TRIANG1D *triang1d;

Reset triang1d ->number_of_points to zero (only needed in ->universal mode).

5.2.2 Bnd-Curve

In the class Bnd_Curve exist additional instance variables to allow points to lie on an associated geometric surface which is connected to this structure via the Triang2d pointer geom. The value vertex[i] is the index of a point in the Triang2d structure (5.3.1).

So changes to the Triang2d structure change also the curve. Such structures are e.g. used as boundary curves: the indices of the boundary vertices in the Triang2d structure are assigned to bnd_curve->vertex. The explicit vertex coordinates bnd_curve->x/y/z allow to modify the curve independently of the surface.

For boundary curves the instance variable neighbour contains the index of triangles along the curve. This information is used e.g. when two Triang2d structures are merged and they have a common boundary curve. Then the new connectivity can be build up easily.

```
typedef struct bnd_curve {
    /* SYSTEM */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* TRIANG1D */
    int number_of_points;
    int max_number_of_points;
    float *x, *y, *z;
    /* BND_CURVE */
    int flag;       /* Flag if bnd_curve ACTIVE */
    TRIANG2D *geom; /* Pointer to triang2d to which the bnd_curve belongs */
    int *vertex;    /* Index of corresponding point in triang2d */
    int *neighbour; /* Index of triangle which the next edge belongs to */
    struct {
        void (*f)(struct bnd_curve *);
        int size;
        double *data;
    } restrict;    /* restriction information for curve - private */
} BND_CURVE;
```

5.2.2.1 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. All other memory (vertex lists) have to be allocated separately using "list-alloc".

GRAPE(bnd_curve, "delete")()

BND_CURVE *bnd_curve;

Delete a Bnd_Curve instance. If bnd ->refcount is not zero it is decremented otherwise all lists and the structure itself is deleted.

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.

GRAPE(bnd_curve, "list-shrink-send")()

BND_CURVE *bnd_curve;

Free unused space of the bnd_curve lists: new lists for bnd_curve ->number_of_points points are allocated, the points with index from 0 to bnd_curve ->number_of_points are copied and the old lists are freed. This method is called by

GRAPE(bnd_curve, "optimize-send")()

BND_CURVE *bnd_curve;

Free unused space of the lists.

GRAPE(bnd_curve, "hardcopy")()

BND_CURVE *bnd_curve;

Create an identical copy of bnd_curve which has its own lists (all data is copied). Of course the geom instance is only ASSIGNed to the copy and not copied (the new instance is associated with the same Triang2d instance as bnd_curve). The pointer to the new Bnd_Curve instance is returned.

GRAPE(bnd_curve, "softcopy")(bnd_copy)

BND_CURVE *bnd_curve;
BND_CURVE *bnd_copy;

Make a softcopy of a Bnd_Curve instance: like "hardcopy" but only the used data (up to number_of_points) is copied. If bnd_copy is not NULL then this instance is used to store the copy otherwise a new Bnd_Curve instance is created.

(int *)GRAPE(bnd_one, "merge")(bnd_two)

BND_CURVE *bnd_one;
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.

5.2.2.2 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.

5.2.2.3 Other Methods on Bnd-Curve

GRAPE(bnd_curve, "interpol")(bnd_1, bnd_2, factor)

GRAPE(bnd_curve, "interpol-fac")(bnd_1, bnd_2, fac1, fac2)

BND_CURVE *bnd_curve;
BND_CURVE *bnd_1, *bnd_2;
double factor, fac1, fac2;

Same as "interpol"/"interpol-fac" on Triang1d (5.2.1.4). 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 bnd_1.

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.

GRAPE(bnd_curve, "assign-curve-send")()

BND_CURVE *bnd_curve;

Assign the bnd_curve ->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.

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

BND_CURVE *bnd_curve;

Print the data of a Bnd_Curve instance to stdout.

5.2.3 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]].

```
typedef struct Cons {
    /* SYSTEM */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* CONS */
    TRIANG2D *geom;            /* Pointer to triang2d to which cons belongs */
```

```
    int number_of_points;      /* Number of points */
    int max_number_of_points;  /* Maximum number of points */
    int *vertex;               /* Indices of the points in triang2d */
    int flag;                  /* Flag for ACTIVE, DISPLAY etc */
} CONS;
```

5.2.3.1 Memory Methods on Cons

GRAPE(cons, "delete")()

CONS *cons;

GRAPE(cons, "list-alloc")(max_points)

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

CONS *cons;
int max_points;

GRAPE(cons, "list_shrink_send")()

CONS *cons;

GRAPE(cons, "optimize-send")()

CONS *cons;

GRAPE(cons, "hardcopy")()

CONS *cons;

GRAPE(cons, "softcopy")(cons_new)

CONS *cons ;
CONS *cons_new;

GRAPE(cons, "merge")(cons-two)

CONS *cons;
CONS *cons-two;

5.2.3.2 Display Methods on Cons

GRAPE(cons, "display")()

GRAPE(cons, "tube")(loc-diskr, loc-scal)

CONS *cons;
int loc-diskr;
float loc-scal;

5.2.3.3 Other Methods on Cons

GRAPE(cons, "get-triang")(triang)

CONS *cons
TRIANG1D triang;

Create a Triang1d instance if triang is NULL and assign the x/y/z entries of the curve from the corresponding entries in cons ->geom ->x/y/z to triang ->x/y/z. The indices for geom ->x/y/z are taken from cons ->vertex.

GRAPE(cons, "verify-use")()

CONS *cons;

After having closed a Triang2d instance, one of the two different parameter lines lying on the same boundary curve points to no longer used coordinates. These lines are no longer used. "verify-use" checks whether a line is still in use. **Definition**: A line is in use, if at least one vertex is used by its geom structure. This definition makes sense, since end points of lines may be out of use after Triang2d "close". For them we have to find points which are close to the original.

GRAPE(cons, "interpol")(cons1, cons2, factor)

GRAPE(cons, "interpol-fac")(cons1, cons1, fac1, fac2)

CONS *cons;
CONS *cons1, *cons2;
double factor, fac1, fac2;

GRAPE(cons, "reflect")(trfmat, orient)

CONS *cons;
MATRIX44 trfmat;
int orient;

GRAPE(cons, "close-send")()

CONS *cons;

GRAPE(cons, "reset")()

CONS *cons;

GRAPE(cons, "follow")(follow, pop-push)

CONS *cons;
INSTANCE *follow[3][3];
char pop-push[3][3];

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

CONS *cons;
LIST *list;

5.3 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 contains information only about the coordinates, the connectivity and neighbourhood relationship between triangles in case of regular triangulations.

The subclass Fe2d contains additionally necessary information for working with finite element solutions of partial differential equations. The function f returns in its parameter vector value[dim], 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. This class contains additional information about the normal vector of each vertex, RGB color vector for each vertex or each triangle, or specify boundary curves. The boundary curves are also necessary for time dependent surfaces that shall be reflected, see e.g. the description of the method "reflect" for TimeObject. Additionally the structure may contain lines on the surface, for example parameter lines. This is controlled by the hor and ver components. The instance variables of a Geom2d object are all optional and are only allocated on request. This reduces the amount of used memory while allowing the class to have the most functionality of the above classes.

5.3.1 Triang2d

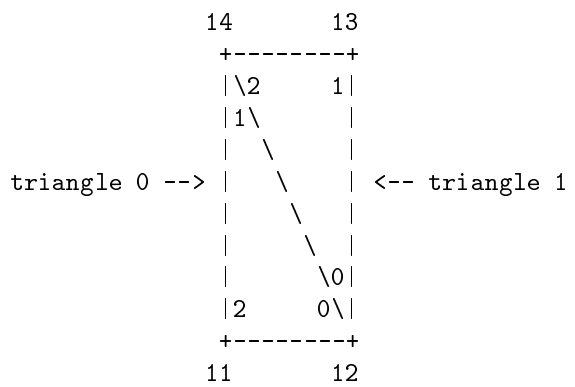
```
typedef struct triang2d {
    /* SYSTEM ONLY */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* TRIANG2D */
    int number_of_points;
    int max_number_of_points;
    float *x, *y, *z;    /* point coordinates, z==NIL possible */
    int number_of_elements;
    int max_number_of_elements;
    INT3 *vertex;    /* vertex indices for all triangles */
    INT3 *neighbour; /* indices of neighbour triangles */
                     /* neighbour[j][i] opposite vertex[j][i] */
 } TRIANG2D;


typedef INT3 int[3];
```

All instance variables based on the Triang classes have a field of 3D vectors in common, which in turn define the location of points in three space. For all Triang classes it is possible to restrict the coordinates to two dimensions by setting the third coordinate pointer z to NULL, those methods that need a third coordinate (like "display") will use zero as default. Be careful when using this feature with subclasses like Geom2d, some methods may not be able to handle this NULL pointer.

For domains with dimension two and three the variable vertex points to an array which contains the numbers of global vertices to each local vertex of all triangles or tetrahedron. For each element the variable neighbour contains information about the neighbour elements.

Example:



The local vertices of each triangle are numbered 0, 1 and 2. The numbers 11, 12, 13 and 14 are the numbers of global vertices corresponding to the local ones. The introduced variables have the following values for this configuration:

```
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
```

Negative values mark boundary edges or surfaces.

5.3.1.1 Memory Methods on Triang2d

GRAPE(triang2d, "delete")()

TRIANG2D *triang2d;

The associated memory of the instance variables is freed and the instance is deleted. This only takes place when triang2d ->refcount is zero, otherwise the reference counter is decreased by one. For a more detailed description of this method see the description of the "delete" method on other classes.

GRAPE(triang2d, "list-alloc")(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.

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-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.

5.3.1.2 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 */
float (*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(x, y, z, var)
    float 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 for interactive use:

```
float plane(x, y, z, var)
    float x, y, z;
    PLANE-PARM *var;

typedef struct plane-parm-type {
    float n-x, n-y, n-z;
    float distance;
} PLANE-PARM; /* Hesse Form */

float ball(x, y, z, var)
    float x, y, z;
    BALL-PARM *var;

typedef struct ball-parm-type {
    float center-x, center-y, center-z;
    float 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;
float 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".

5.3.1.3 I/O Methods on Triang2d

(TRIANG2D *)GRAPE(Triang2d, "read")()

CLASS *Triang2d;

A Triang2d instance is read from the standard textdevice. If "read" fails it returns NULL.

GRAPE(triang2d, "write")()

TRIANG2D *triang2d;

An Triang2d instance is written on the standard textdevice. The output format is described in chapter ->Grape–Files. A NULL pointer is returned if an error occurs.

5.3.1.4 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(triang2d, "interpol")(t2_one, t2_two, factor)

GRAPE(triang2d, "interpol-fac")(t2-one, t2-two, fac1, fac2)

TRIANG2D *triang2d;
TRIANG2D *t2_one, *t2_two;
double factor, fac1, fac2;

Linear interpolation of the two Triang2d instances t2_one and t2_two with factor:

```
triang2d = (1 - factor) * t2_one + factor * t2_two,
```

or with factors fac1 and fac2:

```
triang2d = fac1 * t2_one + fac2 * t2_two.
```

If necessary new vertex lists are allocated for triang2d.
The two Triang2d instances are interpolated pointwise, therefore they should have the same number_of_points and the same connectivity. vertex and neighbour are copied from t2_one.

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;
float 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 8.2.2.1.

GRAPE(triang, "find-pick-intersec")(v, u, w, element, bary)

TRIANG2D *triang;
VEC3 v ; /* point, which is searched */
float 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;
float *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.

The following method creates the field tr2->neighbour, which contains the neighbourhood-relationship of triangles. If not yet existent the memory is allocated.

(TRIANG2D *)GRAPE(tr2,"identify-nbs-send")()

```
TRIANG3D *tr2;
```

GRAPE(Triang2d, "explicit")(name, x, y, z, ax, ex, ay, ey, dx, dy)

```
char *name ; /* name of instance */
float *x, *y, *z; /* pointer to functions of coordinates */
float ax, ex; /* minimum and maximum of the first parameter */
float 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* 7.2.5, 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 */
float 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)();
float 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;
float 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.

The following 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'.

(TRIANG2D *)GRAPE(tr2,"orient-send")()

(TRIANG2D *)GRAPE(tr2,"inverse-orient-send")()

TRIANG2D *tr2;

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")()

TRIANG2D *tr2d;

"identify-nbs-send" builds neighbourhood information for an arbitrary tr2d only out of tr2d ->vertex information. 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\verb1->1num_of_objects = triang\verb1->1number_of_points;
loi\verb1->1objects[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];
```

5.3.2 Fe2d

```
typedef struct fe2d {
    /* SYSTEM ONLY */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* TRIANG2D */
    int number_of_points;
    int max_number_of_points;
    float *x, *y, *z;
    int number_of_elements;
    int max_number_of_elements;
    INT3 *vertex;
    INT3 *neighbour;
    /* FE2D */
    int dimension_of_value; /* 1==scalar, >1 => vector function     */
    int polynomial_order;   /* 1==linear, 2==quadratic,... elements */
    void (*f)();            /* void *f(FE2D fe, int triangle,
                                        VEC3 barycentric-coordinates,
                                        float value[dim])            */
    /* optional */
    int size_of_data;       /* size of user data */
    char *data;             /* and pointer to it */
} FE2D;
```

5.3.2.1 Memory Methods on Fe2d

GRAPE(fe2d, "delete")()

FE2D *fe2d;

Free the memory lists and the userdata of the Fe2d instance and delete the instance itself. This only takes place when fe2d ->refcount is zero otherwise the reference counter is decremented.

5.3.2.2 Display Methods on Fe2d

GRAPE(fe, "isoline-disp")()

GRAPE(fe, "isoline")(select, funct, line_scal, color_scal, offset_scal, high_scal, max, min)

FE2D *fe;
int select ; /* = 1 for "isoline-select", else 0 */ void (*funct)() ; /* function specifying the level lines */
float line_scal ; /* density of lines */
float color_scal ; /* line color */
float offset_scal ; /* offset */
float high_scal, 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;
void (*funct)();
float line, color, offset, high, max, min;

Those particular values which are stored in the array colorbar->values (see section 5.7) 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(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;
float length , mesh, dist;
float 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).

5.3.2.3 I/O Methods on Fe2d

(FE2D *)GRAPE(Fe2d, "read")()

GRAPE(fe2d, "write")()

```
CLASS *Fe2d;
FE2D *fe2d;
```

The methods for reading, and writing a Triang2d instance are extended because of the additional data of the subclass. The corresponding finite element function is treated as linear. This situation can, after writing and reading commands have been given, 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 Triang2d part of an earlier written Fe2d instance. According to this concept see ->last chapters to read and write.

5.3.2.4 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 */
float 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 (5.3.1.4).

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;
float factor;
```

Same as "interpol" on Triang2d (5.3.1.4). 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.

5.3.3 Adapt2d

```
typedef struct adapt2d {
    /* SYSTEM ONLY */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* TRIANG2D */
    int number_of_points;
    int max_number_of_points;
    float *x, *y, *z;
    int number_of_elements;
    int max_number_of_elements;
    INT3 *vertex;
    INT3 *neighbour;
    /* FE2D */
    int dimension_of_value;
    int polynomial_order;
    void (*f)();
    int size_of_data;
    char *data;
    /* ADAPT2D */
    REFINE2D *refine;
    int number_of_bnd_functions;
    int max_number_of_bnd_functions;
    BND_FUNCTION *bnd_function;
    void (*interpol)(), (*restrict)();
    void (*element_compress)(), (*point_compress)();
    } ADAPT2D;


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 {
    float (*g)();
    void *par;
      /* bnd_function[-neighbour[i][j]\verb1->1g(VEC3 x, bnd_function\verb1->1par) = 0.0
         is the boundary with x */
} BND_FUNCTION;
```

5.3.3.1 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.

GRAPE(ad2d, "reduce-mem-send")()

ADAPT2D *ad2d;

Free unused space of all adapt2d data list, usually the difference between number_of_points/elements and max_number_of_points/elements.

5.3.3.2 Display Methods on Adapt2d

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.

5.3.3.3 I/O Methods on Adapt2d

(ADAPT2D *)GRAPE(Adapt2d, "read")()

GRAPE(ad2d, "write")()

CLASS *Adapt2d;
ADAPT2D *ad2d;

The methods for reading, and writing an Fe2d instance are extended because of the additional data of the subclass. The refine field is additional written on the standard device. If one (or more) of the implemented clipping functions (see 5.3.1.2) is used as boundary–function a flag is stored to get the same boundary–function after reading the Adapt2d instance.

5.3.3.4 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-send")()

ADAPT2D *ad2d;

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 5.3.3). 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.

"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.

If boundary segments are refined which belong to a boundary curve stored in the structure BND_FUNCTION, "fit–bndfunctions–send" (see 5.3.3.4) 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.

5.3.4 Geom2d

```
typedef struct geom2d {
  /* superclass structure (ADAPT2D) */
  ...
  /* GEOM2D */
  int flag;   /* Flag for active, display, user normals */
              /* 0x1  GEOM2D_ACTIVE   don't modify geom if FALSE */
              /* 0x2  GEOM2D_DISPLAY  don't draw geom if FALSE */
              /* 0x4  GEOM2D_USERNORM don't compute 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;            /* Pointer to the normals
                              (optional, depends on the flag) */
  VEC3 *color_vertex;      /* Pointer to vertex colors (optional) */
  VEC3 *color_patch;       /* Pointer to patch colors (optional)  */
  LIST_OF_INST *bnd_curve; /* List of geom2d's boundary curves */
  CHAIN *hor, *ver;        /* Linked lists of geom2d's parameter lines */
} GEOM2D;
```

5.3.4.1 Memory Methods on Geom2d

(GEOM2D *)GRAPE(Geom2d, "new-instance")(name)

GRAPE(geom2d, "delete")()

CLASS *Geom2d; GEOM2D *geom2d;
char *name;

"new-instance" creates an instance of class Geom2d with name name. Only the struct itself is allocated and the normal flag is initialized. All other memory has to be allocated separately using "list-alloc". The user can supply its own normals, vertex colors or patch colors. They will be used by "display". If the user wants to use his own normals he has to set geom2d ->normal_flag to G_USER_SUPPLIED.

"delete" deletes an instance of class Geom2d. All data is freed using "list-free".

GRAPE(geom2d, "list-alloc")(mnop, mnoe, mnob, normal, color, boundary)

GRAPE(geom2d, "list-free")()

GRAPE(geom2d, "point-list-alloc")(mnop, tri-connect)

GRAPE(geom2d, "point-list-free")()

GEOM2D *geom2d;
int mnop, mnoe, mnob;
int normal, color, boundary;
GEOM2D *tri-connect;

"list-alloc" allocates all memory lists for a Geom2d instance: memory is allocated for mnop points and mnoe elements, memory for normals, patch and vertex colors and boundaries is allocated if the flags normal/color/boundary are set to YES. If boundary == YES then memory for mnob boundaries is allocated.

"list-free" deletes all lists of the Geom2d instance.

"point-list-alloc" allocates the memory for all point entries (i.e. x, y, z, normal, and color-vertex). If a Geom2d tri-connect is specified then the connectivity from this Geom2d is used.

"point-list-free" deletes all point dependent data.

GRAPE(geom2d, "optimize-send")()

GEOM2D *geom2d;

Free unused space of all geom2d data list, usually the difference between number-of-point/elements and max-number-of-points/elements.

GRAPE(geom2d, "sort-points-send")()

GEOM2D *geom2d;

This method sorts points so that all used points are at the beginning of the point list.

GRAPE (geom2d, "list-shrink-send") ()

GEOM2D *geom2d;

"list-shrink-send" frees unused space of all geom2d data list, usually the difference between number_of_points/elements and max_number_points/elements. Call "sort-points-send" before calling this method to have all used points at the beginning of the point list.

GRAPE(geom2d, "hardcopy")()

GEOM2D *geom2d;

Create an identical copy of geom2d which has its own lists (all lists are copied and the instance entries bnd-curve, hor and ver are hardcopied). The pointer to the new Geom2d instance is returned.

GRAPE(geom2d, "softcopy")(g2-copy)

GEOM2D *geom2d;
GEOM2D *g2-copy;

Make a softcopy of a Geom2d instance: like "hardcopy" but only the used data (up to number_of_points and number_of_elements) is copied. All instance entries (bnd_curve, hor and ver) are softcopied. If g2-copy is not NULL then this instance is used to store the copy otherwise a new Geom2d instance is created.

(int *)GRAPE(geom-one, "merge")(geom-two)

GEOM2D *geom-one;
GEOM2D *geom-two;

Merge two Geom2d instances geom-one and geom-two into geom-one. Allocate new storage if necessary. All vectors of geom-two are appended at geom-one. If the method fails it returns NULL.

5.3.4.2 Display Methods on Geom2d

GRAPE(geom2d, "display")()

GEOM2D *geom2d;

Display a Geom2d instance on the standard device as a patch or as a grid model. If geom2d is displayed in patch mode and geom2d ->normal_flag == G_SYSTEM_SUPPLIED or geom2d ->normal == NULL then normals are computed sending "compute-normals" to geom2d. Also depth cueing is switched off in patch mode.

GRAPE(geom2d, "normals-disp")()

GRAPE(geom2d, "normals")(length)

GEOM2D *geom2d;
float length;

"normals-disp" is the interactive routine for displaying the normals of a Geom2d instance. It adds a ruler to the manager which allows the user to change the length of the normals which are displayed as lines using "normals".

GRAPE(geom2d, "bnd-disp")()

GEOM2D *geom2d;

Display the boundaries (geom2d ->bnd-curve (LIST_OF_INST of Bnd_Curve)) of a Geom2d instance as pipes.

5.3.4.3 I/O Methods on Geom2d

GRAPE(geom2d, "show-send")()

GEOM2D *geom2d;

Print the data of a Geom2d instance to stdout.

5.3.4.4 Other Methods on Geom2d

GRAPE(geom2d, "interpol")(geom-1, geom-2, factor)

GRAPE(geom2d, "interpol-fac")(geom-1, geom-2, fac1, fac2)

GEOM2D *geom2d;
GEOM2D *geom-1, *geom-2;
float factor, fac1, fac2;

Same as "interpol"/"interpol-fac" on the superclasses Fe2d and Triang2d 5.3.1.4. Additionally all special Geom2d components (normals, colors, boundary curves and parameter lines) are interpolated too if present.

GRAPE(geom2d, "reflect")(trfmat, orient)

GEOM2D *geom2d;
MATRIX44 trfmat;
int orient;

Create a new instance of GEOM2D which points are transformed with trfmat. The triangle orientation is changed if orient < 0, if abs(orient) > 1 the normal orientation is also changed. If trfmat == NULL this method has the same effect as "hardcopy".

GRAPE(geom2d, "compute-normals-send")()

GEOM2D *geom2d;

Compute vertex normals for a Geom2d instance. If geom2d ->normal == NULL then the needed memory is allocated.

GRAPE(geom2d, "get-info")(xlist)

GEOM2D *geom2d;
XLIST *xlist;

Create info list entries for an instance of class Geom2d. Entries are class, name, number of points, max number of points, number of triangles, max number of triangles, flags for normals, color and bnd_curve.

(int *)GRAPE(geom2d, "get-bnd-index")(point, dist)

GEOM2D *geom2d;
VEC3 point;
float dist;

Search boundary component in geom2d ->bnd_curve which has a distance <= dist from point to point. Normally used after a "find-pick" and "find-pick-index" call which returns a point picked with the mouse.

If point belongs to an active boundary then the index of this boundary is returned otherwise NULL is returned.

GRAPE(geom2d, "assign-bnd-send")()

GEOM2D *geom2d;

Call "assign-curve-send" 5.2.2.3 on all active boundaries in geom2d ->bnd_curve, i.e. update the boundary curve coordinates using bnd_curve ->geom ->x/y/z and bnd_curve ->vertex.

GRAPE(geom2d, "close-send")()

GEOM2D *geom2d;

Closes the boundaries in a Geom2d instance if possible.

GRAPE(geom2d, "close-dist-send")()

GEOM2D *geom2d;

This method inserts a ruler in an options menu to define the maximal distance to close a boundary.

GRAPE(geom2d, "make-bnd-send")()

GEOM2D *geom2d;

Compute the boundary of a Geom2d instance and the connectivity along the boundary, i.e. adjacent triangles etc. The neighbour field indicates boundary edges. These edges are collected in a list, each edge represented by its two vertex indices, adjacent triangle etc. Then we look for "real" vertices, i.e. vertices at the boundary where the angle between the edge is small enough or where more than two edges meet. From these "real" vertices we start building up the boundary curves out of edge list.

GRAPE(geom2d, "skip-bnd-send")()

GEOM2D *geom2d;

Remove all inactive boundaries from the geom2d ->bnd_curve list and delete them.

GRAPE(geom2d, "get-vertex-wnormals-send")()

GEOM2D *geom2d;

Compute weighted normals for a Geom2d instance using "get-vertex-wnormals" on Triang2d. The computed normals are stored in geom2d ->normal and geom2d ->normal_flag is set to G_USER_SUPPLIED.

GRAPE(geom2d, "refine-local-send")()

GRAPE(geom2d, "coarsen-global-send")()

GEOM2D *geom2d;

First the corresponding methods on the superclass Adapt2d are called and then new boundaries are computed using "make-bnd-send". If geom2d is refined then the new boundary points are corrected according to the boundary curve specification (this does not work if a completely new boundary curve was allocated by "make-bnd-send").

GRAPE(geom2d, "copy-refine")(geom)

GEOM2D *geom2d;
GEOM2D *geom;

Copy the refine vector of geom to geom2d.

5.4 Volumes and 3D Finite Elements

We have introduced the class Triang3d in order to describe the shapes of domains in the dimension three.

```
typedef struct triang3d {
        /* SYSTEM ONLY */
        CLASS *class;
        char *name;
        INSTANCE *next;
        unsigned int refcount;
        /* USER */
        int number_of_points;
        int max_number_of_points;
        float *x, *y, *z;  /* vector of point coordinates */
        int number_of_elements;
        int max_number_of_elements;
        INT4 *vertex;      /* vertex indices for all tetrahedron */
        INT4 *neighbour;   /* neighbour tetrahedron */
        } TRIANG3D;
```

In order to employ finite elements a subclass Fe3d of Triang3d need to be introduced.

```
typedef struct fe3d {
        /* SYSTEM ONLY */
        CLASS *class;
        char *name;
        INSTANCE *next;
        unsigned int refcount;
        /* SUPERCLASS */
        int number_of_points;
        int max_number_of_points;
        float *x, *y, *z;  /* vector of point coordinates */
        int number_of_elements;
        int max_number_of_elements;
        INT4 *vertex;      /* vertex indices for all tetrahedron */
        INT4 *neighbour;   /* neighbour tetrahedron */
        /* SUBCLASS */
        int dimension_of_value; /* 1==scalar, >1 => vector function */
        int polynomial_order; /* 1==linear, 2==quadratic,... elements */
        void (*f)();       /* void *f(FE3D *fe;
                                      int tetrahedron;
                                      VEC4 barycentric_coordinates;
```

```
                                        float value[dim]; )  */
/* only for the users purposes */
int size_of_data;   /* size of user data */
char *data;         /* and pointer to it */
        } FE3D;
```

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.)

5.4.1 Triang3d

5.4.1.1 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.

5.4.1.2 The Methods Read, Write and Delete

GRAPE(triang3d, "write")()

TRIANG3D *triang3d;

An instance is written on the standard text device. The output format is described in chapter "Grape-Files".

A NIL pointer is returned if an error occurs.

(TRIANG3D *)GRAPE(Triang3d, "read")()

An instance of class Triang3d is read from the standard text device. A NIL instance is given back if an error occurs.

GRAPE(triang3d, "delete")()

TRIANG3D *triang3d;

The associated memory of the instance variables is freed-up and the instance are deleted. This only takes place when t->refcount is zero, otherwise the refcounter is decreased by one. For a more detailed description of this event see the description of the "delete" method on other classes.

These three methods are also applicable to the class Fe3d.

GRAPE(fe3d, "write")()

(FE3D *)GRAPE(Fe3d, "read")()

GRAPE(fe3d, "delete")()

The methods for reading, writing, 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. According to this concept see last chapters to read and write.

5.4.1.3 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.

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)

float 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;
float u,w ;
int *element;
float[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;
float (*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)

float 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:

```
float plane(x,y,z,var)
      float x,y,z;
      PLANE_PARM *var;

(PLANE_PARM = { float n_x,n_y,n_z,distance;} [Hesse Form])

float ball(x,y,z,var)
      float x,y,z;
      BALL_PARM *var;

(BALL_PARM = { float 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)

float *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)
float *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;
float 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.

5.4.1.4 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 5.3.1.4).

5.4.2 Fe3d

5.4.2.1 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;
float (*f)();
char *var;
int (*draw_clip)();
char *scal;
int clip_flag;
float 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 */
        float bary[5][4];   /* barycentric coordinates of the */
                            /* intersection points */
        float (*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;
float (*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)();
float 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 float (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;
float *x,*y,*z;
```

int x_n,y_n,z_n;
float *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.

(float *) 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 float field *grad[k+3*j+3*dim*i]* of size *sizeof(float)*3*dim*mnp*.

$$f^j_{,k}(p_i) = \text{grad}\,[\,k + 3\,j + 3\,dim\,i\,]$$

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, float*, float*); */
float 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.

5.5 Moving and Tracing

5.5.1 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 the previous section and in the GRAPE documentation. 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.

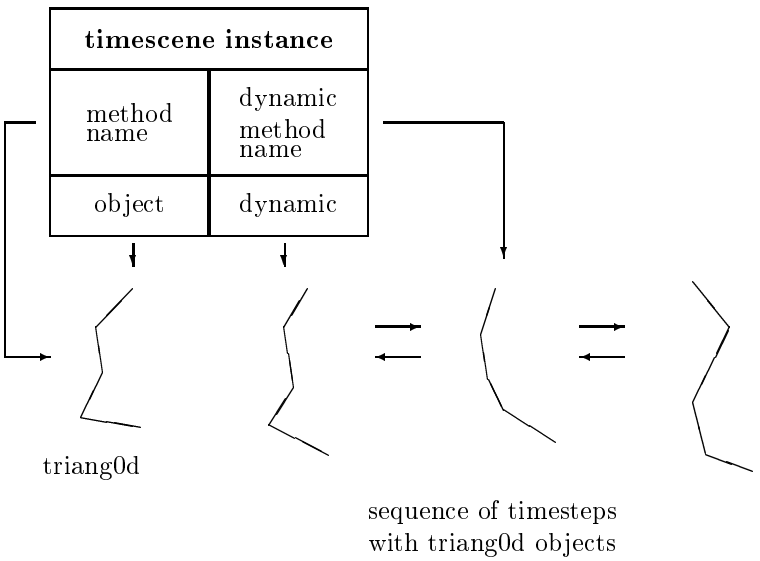


Figure 5.1: example

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 $S^{\delta_h}_{post}(t) = S^{\delta_h}$.

$$\begin{aligned} S^{\delta_h}_{pre}(t+\Delta t) &= \text{OneStep}(v, S^{\delta_h}_{post}(t)) \\ S^{\delta_h}_{post}(t+\Delta t) &= \text{AdaptMesh}(S^{\delta_h}_{pre}(t+\Delta t) \end{aligned}$$

Here **OneStep** is one iteration of the described ODE scheme on the vertex points based on the modified **EulerCauchy** scheme.

$$x(t+\delta_t) = \text{EulerCauchy}(v, t, \delta_t, x(t)) \quad \forall_{x(t)} x(t) \text{ vertex in } S^{\delta_h}_{post}(t)$$

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.

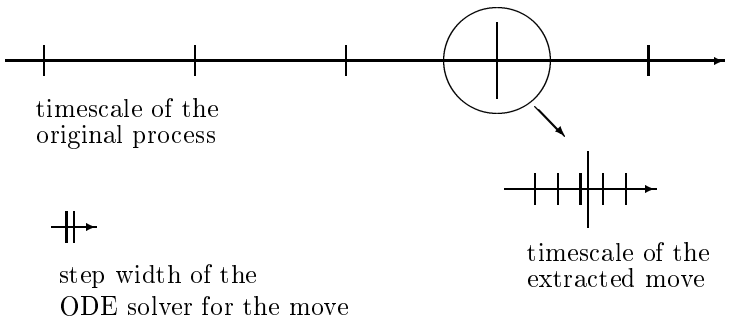


Figure 5.2: the different scales

5.5.2 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 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.

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 7.1.1) *move-trace* contains a

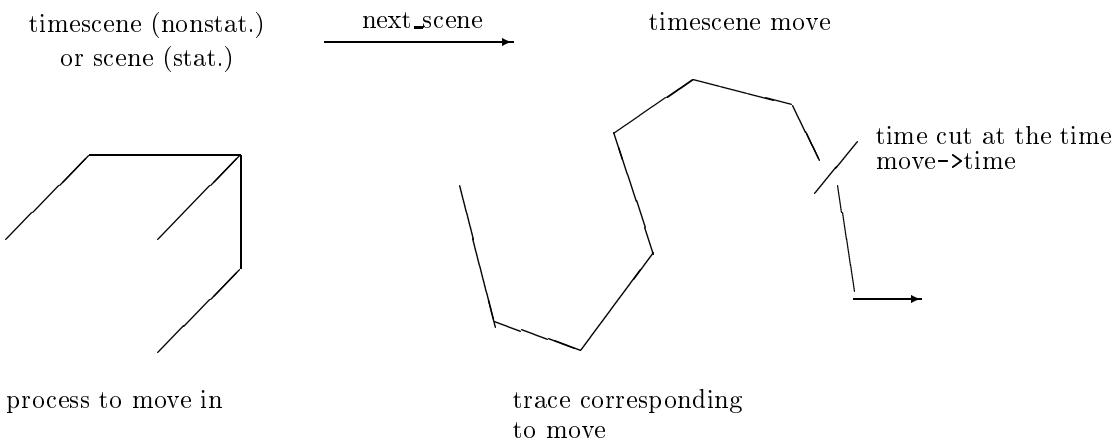


Figure 5.3: moving process

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*.

```
typedef struct triang0d {
  /* SYSTEM ONLY */
  CLASS *class;
  char *name;
  INSTANCE *next;
  unsigned int refcount;
  /* TRIANG1D */
  float x,y,z;
} TRIANG0D;
```

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.

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

```
typedef struct trace {
  /* SYSTEM ONLY */
  CLASS *class;
  char *name;
  INSTANCE *next;
  unsigned int refcount;
  /* TRIANG1D */
  int number_of_points;
  int max_number_of_points;
  float *x, *y, *z;
  /* SUBCLASS TRACE */
  float *t;
} TRACE;
```

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, "write")()

(TRACE *)GRAPE(trace, "read")()

(TRACE *)GRAPE(Trace, "read")()

TRACE *trace
CLASS *Trace

These methods write (read) a trace instance to (from) the standard text device and return the written (read) trace (or NULL which indicates failure). If "read" is sent to the class, the data is read into a new instance, otherwise into the receiver *trace*.

GRAPE(trace, "shorting")(eps)

TRACE *trace
float 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.

5.5.3 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
float 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.

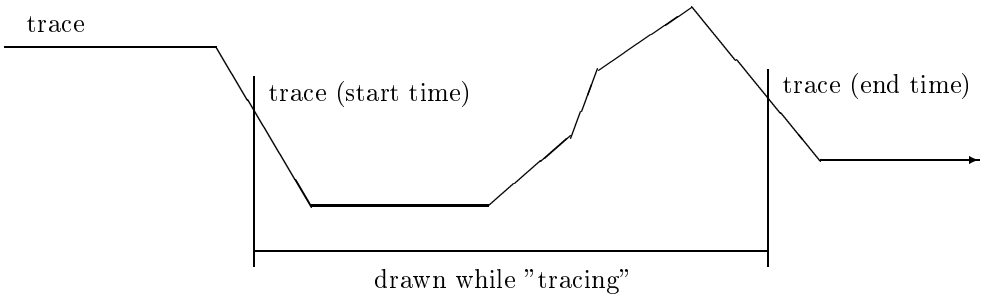


Figure 5.4: tracing

GRAPE(timestep, "tracing")(start_time,end_time)

GRAPE(timestep, "tracing-disp")()

TIMESTEP *timestep
float 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.

5.5.4 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:

```
typedef struct cloud {
  /* SYSTEM ONLY */
  CLASS *class;
```

```
  char *name;
  INSTANCE *next;
  unsigned int refcount;
  /* CLOUD */
  int number_of_points;
  VEC3 *x;
} CLOUD;
```

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" message on scene invokes the sending of the "2cloud" message to the scene object and replaces the latter by the generated cloud.

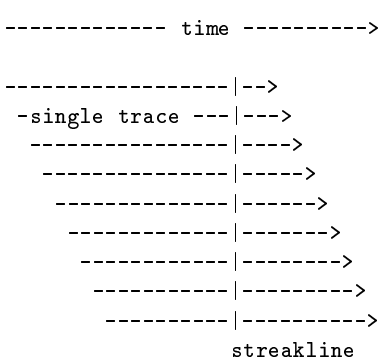
5.5.5 Streaklines

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:

```
typedef struct streakline {
  /* SYSTEM ONLY */
  CLASS *class;
  char *name;
  INSTANCE *next;
  unsigned int refcount;
  /* TRIANG1D */
  int number_of_points;
```

```
  int max_number_of_points;
  float *x, *y, *z;
  /* SUBCLASS TRACE */
  float *t;
  /* SUBCLASS STREAKLINE */
  int number_of_steps;
  int max_number_of_steps;
} STREAKLINE;
```

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*.



A *streak time* ruler in the moving projects allows to fix the length of the single traces, where the interval from *start time* to *endtime* determines the total considered time interval. A button *streak moves* 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
float 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.

GRAPE(streak, "get-times")(stime,etime)

STREAKLINE *streak
float *stime,*etime

get-times fills *stime. etime* with the maximal and minimal time of a streakline.

5.6 General Meshes

5.6.1 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:	$\{(\mu_1, \mu_2, \mu_3) : 0 \leq \mu_i \leq 1\}$.

5.6.2 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.

5.6.2.1 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    */
 float     **coord;               /* local coordinates of the vertices  */
 int       parametric_degree;     /* parametric element if > 1          */
 int       (*world_to_coord)(ELEMENT3D *, float *, float *);
```

```
 void      (*coord_to_world)(ELEMENT3D *, float *, float *);
 int       (*check_inside)(ELEMENT3D *, float *);

 ELEMENT3D *(*neighbour)(ELEMENT3D *, int, int, float *, 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 ($0 \leq j <$ `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 ($0 \leq j <$ `number_of_polygons`, $0 \leq i <$ `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]` ($0 \leq j <$ `number_of_polygons`, $0 \leq i <$ `polygon_length[`j`]`).

`coord[`k`][`i`]` is the i–th local coordinate of the k–th vertex from the polyhedron ($0 \leq i <$ `dimension_of_coord`, $0 \leq k <$ `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.

Unstructured grids usually provide such information explicitly.

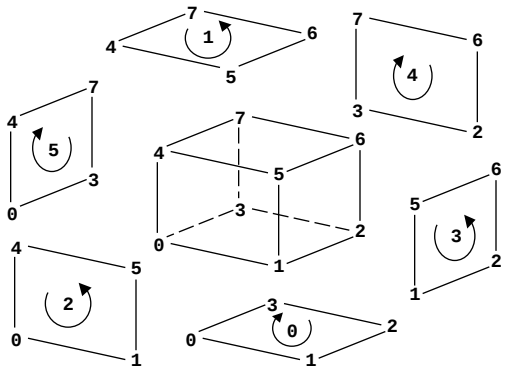


Fig. 1: Prototype of a cube

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, float[], float[]);
  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;
```

5.6.3 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
float (*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 = { float n_x,n_y,n_z,distance;};

*float mplane(VEC3 x, MPLANE_PARM *var)*

typedef struct MBALL_PARM = { float center_x,center_y,center_z,radius;};

*float 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;
  float *coord;
  float (*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
float 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, float *coord, float *val)
float 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, float *coord, float *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.

5.6.4 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]
```

5.6.5 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 float c_c0[3] = {0.,0.,0.}, c_c1[3] = {1.,0.,0.};
static float c_c2[3] = {1.,1.,0.}, c_c3[3] = {0.,1.,0.};
static float c_c4[3] = {0.,0.,1.}, c_c5[3] = {1.,0.,1.};
static float c_c6[3] = {1.,1.,1.}, c_c7[3] = {0.,1.,1.};

static float *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,
                                  float *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, float coord[], float 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.

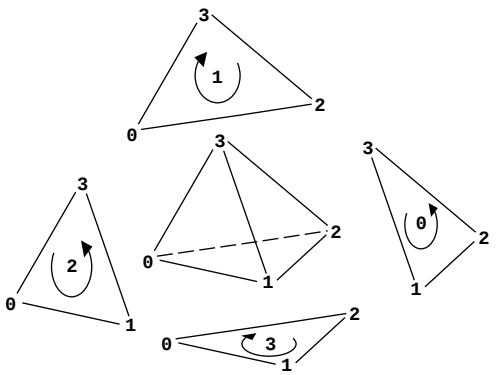
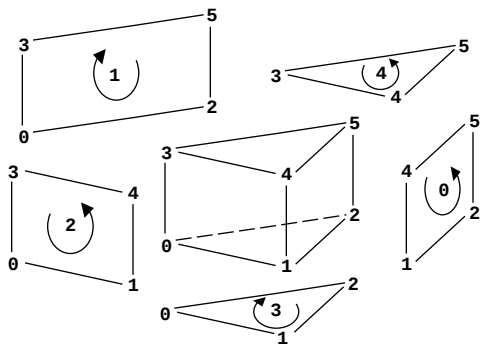


Fig. 2: Prototype of a tetrahedron



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 float p_c0[4] = {1.,0.,0.,0.}, p_c1[4] = {0.,1.,0.,0.};
static float p_c2[4] = {0.,0.,1.,0.}, p_c3[4] = {1.,0.,0.,1.};
static float p_c4[4] = {0.,1.,0.,1.}, p_c5[4] = {0.,0.,1.,1.};
static float t_c0[4] = {1.,0.,0.,0.}, t_c1[4] = {0.,1.,0.,0.};
static float 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 float *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 float *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
};
```

5.6.6 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;
  float     **coord;
  int       parametric_degree;
  int       (*world_to_coord)(ELEMENT2D *,float *, float *);
  void      (*coord_to_world)(ELEMENT2D *,float *, float *);
  int       (*check_inside)(ELEMENT2D *,float *);

  ELEMENT2D *(*neighbour)(ELEMENT2D *, int, int, float *, 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, float[], float[]);
  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;
};
```

5.6.7 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
float 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 5.7). 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
float *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.

5.7 Using the colorbar

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

$$\text{value} = \textit{colorbar}\text{->}\textit{min} + (n + 0.1) * \text{line-distance} \qquad \text{with } n \in \{0, 1, 2, \ldots\}.$$

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.

5.7.1 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;
```

5.7.2 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 6

Supervising Classes

6.1 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 pointers 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 7, 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.

6.1.1 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 float MATRIX44[4][4];


typedef struct suprop {
    ...
    /* superclass structure */
    SUPROP_DEV suprop_dev;
    int grid_patch;
} SUPROP;


typedef struct suprop_dev {
    float emission[3];         /* rgb color emission (incl. intensity) */
    float ambient[3];          /* ambient rgb reflectance */
    float diffuse[3];          /* diffuse rgb reflectance */
    float specular[3];         /* specular rgb reflectance */
    float specular_exp;        /* specular reflectance exponent */
    float transparency;        /* transparency factor [0..1] */
} SUPROP_DEV;
```

6.1.1.1 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, "delete")()

SCENE *scene

This method removes the instance and frees the allocated memory if the reference counter equals zero. Otherwise only the reference counter will be decremented. All instances which belong to this scene will receive a `"delete"` message. To increment the reference counter compare `ASSIGN` in 4.4.5.2.

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
float 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
float 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.

GRAPE(scene, "write")()

(SCENE *)GRAPE(Scene, "read")()

SCENE *scene
CLASS Scene

The method "write" creates an output of this instance on the standard textdevice. Moreover all scene's objects are written on the standard textdevice. If any object exists twice it is written only once.
"read" reads instances of class Scene (written by "write") from the standard textdevice.

6.1.1.2 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.

GRAPE(suprop, "write")()

(SUPROP *)GRAPE(suprop, "read")()

(SUPROP *)GRAPE(Suprop, "read")()

SUPROP *suprop
CLASS *Suprop

The method "write" creates an output of this instance on the standard text device. 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).

(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.

6.1.2 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;
```

6.1.2.1 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.

6.1.2.2 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 object's 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-scene's 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 float *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.

6.1.3 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 */
    float time,object_time;
    INSTANCE *dynamic;
    char *dynamic_method_name;
    int sync;
} TIMESCENE;
```

6.1.3.1 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.

6.1.3.2 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
float 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.

(TIMESCENE *)GRAPE(timescene, "write")()

(TIMESCENE *)GRAPE(timescene, "read")()

(TIMESCENE *)GRAPE(TimeScene, "read")()

TIMESCENE *timescene
CLASS *TimeScene

These methods write (read) a timescene instance to (from) the standard text device and return *NULL* in case of failure. Writing a timescene consists mainly in sending "write" to its dynamic object and "^write" to *timescene* itself (the interpolating timescene object is uncoupled before). If the timescene contains no dynamic instance, though, "write" is directly reached to the next scene.

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).

FunctionRuler

6.2 TimeStep

As described in section 3.5.4 the class TimeStep is one dynamic process which can be used in Grape. It is actually the only dynamic class which is fully implemented and provided with methods. See section 3.5.4 for a sketch of the doubly liked list of key frame objects:

Each step has two pointers to data, one to a pre_object and one to a post_object. Having the same connectivity in the whole sequence both pointers point to the same key object. But in time the connectivity may change. To keep the interpolation mechanism working the two objects at a step with time t where the connectivity changes have to represent the same geometric data but with two different connectivities. The pre_object has the connectivity of the data during the time up to now, and interpolation for times less than t are done using the connectivity of the pre_object. The post_object has respectively the connectivity of the pre_object of the next time step at time t2.

```
typedef struct timestep {
    ...
    /* superclass structure (INSTANCE) */
    float time;
    struct timestep *pre_step,*post_step;
    INSTANCE *pre_object,*post_object;
 } TIMESTEP;
```

6.2.1 General methods on TimeStep

GRAPE(timestep, "delete")()

TIMESTEP *timestep

This method sends the message "delete" to all pre and post_objects and steps and finally deletes the timestep itself.

(TIMESTEP *)GRAPE(timestep, "write")()

(TIMESTEP *)GRAPE(timestep, "read")()

(TIMESTEP *)GRAPE(Timestep, "read")()

TIMESTEP *timestep
CLASS *Timestep

These methods recursively write (read) a timestep list to (from) the standard text device and return the written (read) timestep (NULL in case of failure). No requirements are made as to the position of *timestep* in the list. If "read" is sent to the class, the data is read into a new instance, otherwise into the receiver *timestep*.

(TIMESTEP *)GRAPE(timestep, "hardcopy")()

(TIMESTEP *)GRAPE(timestep, "softcopy")(new_timestep)

TIMESTEP *timestep
TIMESTEP *new_timestep

In the case of the hardcopy message a real copy of the whole timestep structure is created. This includes real copies of all pre_/post_objects. Different to that the softcopy makes a copy of the timestep structure, pointer to pre_/post_objects are copied, NOT the real pre/post_objects, if an object is given it will be recycled.

(TIMESTEP *)GRAPE(TimeStep, "assign-vec")

(vec_of_inst, vec_of_time, n)
INSTANCE **vec_of_inst;
double *vec_of_time;
int n;

This method creates a list of n timesteps. The timestep i contains vec_of_inst[i] as its pre_object and post_object and vec_of_time is its time. The vec_of_time array has to be ordered.

GRAPE(timestep, "universal")(char *method)

TIMESTEP *timestep
char *method

The message method is passed to all pre_- and post_objects.

(INSTANCE *)GRAPE(timestep, "get-object")(inst,time)

TIMESTEP *timestep
INSTANCE *inst
float time

This methods is the central one. It creates (interpolate) a current object for time 'time' from timestep and store it in 'inst'. If inst == NIL it is created.

GRAPE(tst, "put")(pre,post,t)

GRAPE(tst, "get")(t)

TIMESTEP *tst
INSTANCE *pre, *post
float t

The method put adds a timestep with pre and post as its pre_ and post_object and time t to an existing timestep sequence. If tst is a superclass of TimeStep a first TimeStep instance is created and returned. The method get returns the timestep with time t if it exists.

We will now sketch how to map numerical data located on disc onto a TimeScene structure with a list of adaptive timesteps as its dynamic instance variable. Here we will only give a brief example in form of a C like algorithm. Let us suppose that the timestep size is fixed and the data of the presteps and poststeps is stored in a list of files containing the number of the timestep as postfix. Furthermore only the poststep data files contain geometry information.

```
TIMESCENE *tsc;
TIMESTEP *tsp;
INSTANCE *FE,*FEPRE,*FEPOST;
float fact;

char name_pre[100],name_post[100],*path;

path = << path name >>
fact = << timestep size >>

tsc = (TIMESCENE *)GRAPE(TimeScene,"new-instance")(<<name>>);
for(frac_time = start_time;frac_time<=end_time;frac_time++) {

   main_time = fact * (frac_time - start_time);
   sprintf(name_pre,"spre%d.dat",path,frac_time);
   sprintf(name_post,"%spost%d.dat",path,frac_time);
   printf("read %s %s\n",name_pre,name_post);

   if(frac_time == start_time) {
     << read object FE with name name_post >>
     tsc->dynamic = (INSTANCE *)GRAPE(TimeStep,"put")
                                    (NIL,FE,main_time);
```

```
    }
    else {
      if(frac_time == end_time) {
        << read object FEPRE with name name_pre >>
        << copy geometry information from FE to FEPRE >>
        tsc->dynamic = (INSTANCE *)GRAPE(tsc->dynamic,"put")
                                        (FEPRE,NIL,main_time);
      }
      else {
        printf("read %s %s\n",name_pre,name_post);
        << read object FEPRE with name name_pre >>
        << read object FEPOST with name name_post >>
        << copy geometry information from FE to FEPRE >>
        FE = FEPOST;
        tsc->dynamic = (INSTANCE *)GRAPE(tsc->dynamic,"put")
          (FEPRE,FEPOST,main_time);

      }
    }
}
```

6.3 Dynamic Objects Supervising TimeStep

The following classes simplify the use of the class TimeStep for various applications. They free the user from many tasks allowing him to concentrate on the computation to be done. For example they provide all memory management and a set of time dependent integer and double variables.

6.3.1 Time-Object

Time-Object is the simplest class supervising a TimeStep besides TimeScene. The class of the geometry under the timestep may be of arbitrary type. Once this type is declared the user only has to set up a description file with information about the number of frames. Additionally he may ask for a number of integer and double parameters for which the system will install FunctionRulers and read control splines if they exist. The only task for the user is to write a computation routine to get his geometric data.

When loading the project *Time_Object* all buttons necessary to read the description file and execute the computation are installed. See 7.2.1 for a description of the project.

6.3.1.1 The Time-Object Structure

```
typedef struct time_object {
   /* superclass structure (INSTANCE) */
   ...
   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;             /* Timeobject parameter (see
                                                            below) */
```

```
   CHAIN *geometry, *inter;          /* Timesteps with timeobject
                                        geometry                    */
                                     /* Interpol instance for
                                        displays etc                */
   SCENE *tree;
} TIME_OBJECT;


typedef enum {
  tskGeometry,   /* bitwise flags */
  tskGauss,
  tskConjugate,
  tskDomain,
  tskNewton,
  tskExplicit,
  tskAny
} TASK_TYPE;
```

6.3.1.2 Time-Parm

Time_Parm is the structure controlling the parameters of a Time_Object.

```
typedef struct time_parm {
    /* superclass structure (INSTANCE) */
    ...
    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 */
} TIME_PARM;
```

GRAPE(time_parm, "delete")()

TIME_PARM *time_parm

Delete an instance of class Time_Parm. All data is deleted using "list-free", all time_parm rulers are removed from the manager and deleted.

GRAPE(time_parm, "list-alloc")()

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

TIME-PARM *time_parm

"list-alloc" allocates the memory for the integer and double variables and the pointer arrays for the correspondent function rulers. The ruler itself are created when the input file is read.

"list-free" deletes all data including the function rulers (not only the pointers allocated in "list-alloc") which are removed from the manager and deleted.

GRAPE(time_parm, "get-info")(xlist)

TIME-PARM *time_parm
XLIST *xlist

Create info–list entries for an instance of class Time_Parm. Entries are class, name, number of parameters and parameter values.

GRAPE(time_parm, "set")(on_off)

TIME-PARM *time_parm
int on_off

Activate (on_off == ON) or deactivate (on_off == OFF) all rulers belonging to time_parm.

GRAPE(time_parm, "switch-send")()

TIME-PARM *time_parm

Toggle the status off all rulers belonging to time_parm (ON <-> OFF).

6.3.1.3 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. Only the descriptor list_of_task is allocated completely, status and time_step have to be allocated separately using "list-alloc".

GRAPE(time_object, "delete")()

TIME_OBJECT *time_object

Delete an instance of class Time_Object. All data is deleted using "list-free".

GRAPE(time_object, "list-alloc")()

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

TIME_OBJECT *time_object

"list-alloc" allocates and initializes status and time_step for an instance of class Time_Object. num_of_frames has to be initialized before calling this method!

"list-free" deletes all data which has not been allocated by "new-instance" including user allocated geometry, inter and tree instances.

GRAPE(time_object, "hardcopy")()

TIME_OBJECT *time_object

Make a hardcopy of time_object: all variables, lists and instances except for inter and tree are hardcopied to a new Time_Object instance which is returned.

GRAPE(time_object, "softcopy")(to-new)

TIME_OBJECT *time_object, *to-new

Make a softcopy of time_object: like "hardcopy" but the instance entries are softcopied (only the used data is copied). If to-new is not NULL then this instance is used to store the copy otherwise a new Time_Object instance is created. The copied instance (i.e. to-new if it was not NULL) is returned.

GRAPE(time_object, "merge-send")()
TIME_OBJECT *time_object

Merge the time_object's geometry using "merge-send" and deactivate all time_object ->time_parm rulers by sending "set"(OFF).

6.3.1.4 Display Methods on Time-Object

GRAPE(time_object, "display")()

GRAPE(time_object, "geometry-disp")()
TIME_OBJECT *time_object

Display the time section of time_object ->geometry for time time_object ->time: the time section is created by calling "get-object"(time_object ->time) and stored in time_object ->inter which then is displayed.

6.3.1.5 Compute Methods on Time-Object

GRAPE(time_object, "compute-video-send")()
TIME_OBJECT *time_object

Compute all frames for time_object by sending "compute-send" to each frame. The current frame number and time are saved and restored after the computation.

GRAPE(time_object, "compute-send")()
TIME_OBJECT *time_object

This is the method which actually computes the data for Time_Objects. All memory management has to be done here because the class of the time_object geometry is arbitrary. If the user wants to compute his own Time_Objects he can replace this method with his own "compute-send" method.

GRAPE(ruler, "parameter-changed")()
TIME_OBJECT *time_object
RULER *ruler

This is the default action for a parameter ruler for an instance of class Time_Object and some of its subclasses. If a parameter is changed the frame(s) are immediately updated using the new parameter. The instance the ruler belongs to has to be set as the instance of the ruler with `"set-instance"`.

6.3.1.6 I/O Methods on Time-Object

GRAPE(time_object, "read-data")(name)
TIME_OBJECT *time_object
char *name

Read the input file name.to and create the parameter lists and the rulers associated to time_object.

6.3.1.7 Other Methods on Time-Object

GRAPE(time_object, "universal")(method)

TIME_OBJECT *time_object
char *method

Send display methods to time_object ->inter (which is updated first by calling "get-object") and all other methods to time_object ->geometry.

GRAPE(time_object, "follow")(follow, pop_push)

TIME_OBJECT *time_object
INSTANCE *follow[3][3]
char pop_push[3][3]

Set the environment data of the instance time_object for the object hierarchy in the interactive package (manager).

GRAPE(time_object, "get-info")(xlist)

TIME_OBJECT *time_object
XLIST *xlist

Create info list entries for a Time_Object instance. Entries are class, name, refcount, current frame number, number of frames, task and current time and time and status for each frame.

GRAPE(time_object, "reset-send")()

TIME_OBJECT *time_object

Send "reset" to time_object ->geometry and reset the status of all frames.

(double *)GRAPE(Time_Object, "get-time")()

GRAPE(Time_Object, "get-time-ruler")()

GRAPE(time_object, "set-time")(time)

GRAPE(time_object, "update-time")()

CLASS *Time_Object
TIME_OBJECT *time_object
double time

"get-time" returns the pointer to the (static) TIME_ZEIT variable which is used for Time_Objects instances (time_object ->time) and for the standard time ruler rultime.

"get-time-ruler" returns the pointer to the standard time ruler rultime. rultime is created automatically when the project *Time_Object* (or one of its subprojects) is added.

"set-time" sets time_object ->time to time and updates the time ruler.

"update-time" has to be called when the time-objects time is changed. It will update all time dependent parameters and the associated rulers. Normally "set-time" should be used, this will call "update-time" if necessary.

(GROUP *)GRAPE(Time_Object, "get-frame-buttons")()

GRAPE(time_object, "previous-frame")()

GRAPE(time_object, "next-frame")()

GRAPE(time_object, "set-frame")(frame)

GRAPE(time_object, "update-frame")()

CLASS *Time_Object
TIME_OBJECT *time_object
int frame

"get-frame-but" returns the pointer to the group with the standard frame button. The frame buttons are created automatically when the project *Time_Object* (or one of its subprojects) is added.

"previous-frame" decrements the frame number. If no previous frame exists the frame number is set to time_object ->num_of_frames - 1. This is the default action for the frame down button [<-].

"next-frame" increments the frame number. If no next frame exists the frame number is set to zero. This is the default action for the frame up button [->].

"set-frame" sets time_object ->frame to frame and update the frame button.

"update-frame" will print the new frame number on the frame number input field.

(CYCLE_BUTTON *)GRAPE(Time_Object, "get-task-button")()

GRAPE(time_object, "set-task")(task)

CLASS *Time_Object
TIME_OBJECT *time_object
TASK_TYPE task

"get-task-button" returns the pointer to the standard task button. The task button is created automatically when the project *Time_Object* (or one of its subprojects) is added.

"set-task" sets the task for time_object to task and updates the task button.

GRAPE(time_object, "switch-comp-send")()

TIME_OBJECT *time_object

Toggle the status (ON/OFF) of all rulers belonging to time_object by sending "switch-send" to time_object ->time_parm.

6.3.2 Surface

This class is the fundamental subclass of Time_Object when working with time dependent surfaces. The geometry at each time step is assumed to be of class Geom2d. The surfaces may have boundaries and parameter lines which can be supplied inside the Geom2d structure.

Additional to the main geometry further data sets may be stored: a domain, the stereographic projection of the normals onto the plane, i.e. the gauss map, and a second data set with the same connectivity as the geometry. This may be used for different purposes, for example handling the data of a two parameter family (additional to the time parameter) in memory. An intensive use of the timestep conjugate is for example described in the explanations for the classes SurfaceBuild and Amandus.

The different geometries domain, gauss, geometry and conjugate may be computed separately by switching the task button [TASK:...] or by changing the default setting in the description file.

A specialty is that reflection operations may be executed on the geometry and the conjugate geometry at the same time. Also the boundary curves and parameter lines may be reflected as well.

All other new instance variables of the class Surface should not be used by inexperienced users. They are mainly used during reflection operations to store the data hierarchy. Internally the reflected objects are stored in scene hierarchies. The reflected boundary lines and parameter lines may be handled independently of the geometry, therefore they also build up separate scene hierarchies. Experienced user may include this additional scene hierarchies into the main hierarchy

visible in the manager menu [mngr] below the node of Surface by sending the message "make-tree" using the [send method] button. One can then walk in this additional scene hierarchy and set for example other display methods for specific patches to emphasize them.

Up to now no compute routine exists for this class and therefore no description file. On the subclasses of Surface (Explicit, Amandus, etc.) a variety of compute routines exist.

6.3.2.1 The Surface Structure

```
typedef struct surface {
    /* superclass structure (TIME_OBJECT) */
    ...
    CHAIN *model;                    /* This is the model which is
                                        used for almost all opera-
                                        tions (computation, reflec-
                                        tion etc). Depending on the
                                        current TASK, model points
                                        to geometry, 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 */
} SURFACE;

typedef struct reflection_control {
    /* superclass structure (INSTANCE) */
    ...
    int type;   /* use following reflection type if flag is TRUE */
                /* flags have descending priority! */
                        /* 0x1  REFL_ALL        */
                        /* 0x2  REFL_AUTOMATIC  */
                        /* 0x4  REFL_PLANAR     */
                        /* 0x8  REFL_LINE       */
                        /* 0x16 REFL_POINT      */
                        /* 0x32 REFL_NORMAL     */
                        /* 0x64 REFL_CIRCLE     */
    int obj;    /* reflect the following objects if flag is TRUE */
                        /* 0x1  REFL_MODEL      */
                        /* 0x2  REFL_GEOMETRY   */
                        /* 0x4  REFL_DOMAIN     */
                        /* 0x8  REFL_GAUSS      */
                        /* 0x16 REFL_CONJUGATE  */
                        /* 0x32 REFL_NEWTON     */
    int last_obj_index;
    int object_index, bdy_index, bdy2_index;
} REFLECTION_CONTROL;
```

6.3.2.2 Memory Methods on Surface

(SURFACE *)GRAPE(Surface, "new-instance")(name)

CLASS *Surface
char *name

Create an instance of class Surface with name name. First "new-instance" on the superclass Time_Object is called, then all descriptors are allocated. All other instances have to be allocated separately using "list-alloc".

GRAPE(surface, "delete")()

SURFACE *surface

Delete an instance of class Surface. All data is deleted using "list-free".

GRAPE(surface, "list-alloc")(num_points, mnob)

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

SURFACE *surface
int num_points, mnob

"list-alloc" allocates and initializes the Surface instance completely. First "list-alloc" on the superclass Time_Object is called, then for each frame (surface ->num_of_frames has to be initialized!) the required memory for a maximum of num-points points and mnob boundaries for each triangulation is allocated by calling "list-alloc" on Geom2d.

"list-free" deletes all surface instance–entries (model, domain, gauss etc) which have been allocated by "list-alloc" and then calls "list-free" on the superclass Time_Object.

GRAPE(surface, "hardcopy")()

SURFACE *surface

Create an identical copy of surface: all variables, lists and instances but inter, conj-inter and tree are hardcopied to a new Surface instance which is returned.

GRAPE(surface, "softcopy")(sf_new)

SURFACE *surface, *sf_new

Make a softcopy of surface: like "hardcopy" but the instance entries are softcopied (only the used data is copied). If sf-new is not NULL then this instance is used to store the copy otherwise a new Surface instance is created. The new instance (i.e. sf-new if it was not NULL) is returned.

GRAPE(surface, "merge-send")()

SURFACE *surface

Send "merge-send" to all surface's geometries. This is done by sending "merge-send" to domain, gauss and conjugate and then calling "merge-send" on the superclass Time_Object.

6.3.2.3 Display Methods on Surface

GRAPE(surface, "display")()

GRAPE(surface, "domain-disp")()

GRAPE(surface, "gauss-disp")()

GRAPE(surface, "geometry-disp")()

GRAPE(surface, "conjugate-disp")()

SURFACE *surface

Display the time section of surface ->model (surface ->domain etc) for time surface ->time: the time section is created by calling "get-object"(surface ->time) and stored in surface ->inter which then is displayed.

GRAPE(surface, "associate-disp")()

GRAPE(surface, "associate")(asso)

SURFACE *surface
double asso

Display the associate surface for surface ->geometry and surface ->conjugate. "associate-disp" adds the associate ruler to an option menu and then calls "associate" which first creates surface ->inter and surface ->conj-inter by calling "get-object" and then displays asso * surface ->inter + (1 − asso) * surface ->conj_inter.

6.3.2.4 I/O Methods on Surface

(LIST *)GRAPE(surface, "open-in")(list, device, picked_name)

SURFACE *surface
LIST *list
TEXTDEVICE **device
char *picked_name

Create the list of input files for class Surface ('*.sf'). This method is called when the [read *.in] button in the option menu is pressed.

At the moment there are no input files for class Surface. This method is only supplied for consistency. This is also the case for the following method:

GRAPE(surface, "read-data")(name)

SURFACE *surface
char *name

Read the input file name.sf (there are currently no input files for class Surface).

6.3.2.5 Other Methods on Surface

GRAPE(surface, "get-object")(inst, time)

SURFACE *surface
INSTANCE *inst
double time

Interpolate the time section for time time from surface ->model and store it in surface ->inter which is returned.

(CHAIN *)GRAPE(surface, "get-bnd-curve")(bnd_chain)

SURFACE *surface
CHAIN *bnd_chain

Extract the boundaries from surface ->model's Geom2d instances and hardcopy them to bnd_chain which is returned. If bnd_chain is NULL a new Chain instance is created.

GRAPE(surface, "universal")(method)

SURFACE *surface
char *method

Send display methods to surface ->inter (which is updated first by calling "get-object") and all other methods to surface ->model.

GRAPE(surface, "close-send")()

SURFACE *surface

Send "close-send" to all surface's geometries. This is done by sending "close-send" to domain, gauss and conjugate and then calling "close-send" on the superclass Time_Object.

GRAPE(surface, "follow")(follow, pop_push)

SURFACE *surface
INSTANCE *follow[3][3]
char pop_push[3][3]

Set the environment data of the instance surface for the object hierarchy in the interactive package (manager).

GRAPE(surface, "get-info")(xlist)

SURFACE *surface
XLIST *xlist

Create info list entries for a Surface instance. Entries are class, name, refcount, current frame number, number of frames, task, time, associated value, number of points and number of elements of the geometry (if available) and time and status for each frame.

GRAPE(surface, "reset-send")()

SURFACE *surface

Send "reset" to all surface's geometries (geometry, domain, gauss and conjugate) and reset the status of all frames. All reflected instances are deleted.

GRAPE(surface, "debug")()

SURFACE *surface

Debugging for instances of class Surface (and subclasses): points and objects can be picked with the mouse and then information about the points and objects is given (coordinates, index, transformation, etc). Point can be picked with the left mouse button and objects with the right mouse button. The middle button exits this method.

GRAPE(surface, "make-tree-send")()

SURFACE *surface

"make-tree-send" creates a surface ->tree. This methods adds some new timescene nodes to the hierarchy tree which all have the object surface ->model. The timescenes have different display methods which allow the separate manipulation of the geometry, its boundaries and its parameter lines.

GRAPE(surface, "find-pick")(v, u, w, obj, dist)

SURFACE *surface
VEC3 v
float u, w
INSTANCE **obj
float *dist

If surface ->inter exists and is displayed then "find-pick" is send to it and the found point v and object obj are returned. See "find-pick" on Scene (6.1.1.1) or on Triang2d (5.3.1.4) for an explanation.

GRAPE(surface, "reflect-send")()

GRAPE(surface, "reflect")()

SURFACE *surface

"reflect-send" is the default action for the [reflect] button in project *Surface*. The user can interactively pick the boundary for the reflection and the objects to reflect. After this has been done "reflect" is called to do the reflection.

GRAPE(surface, "reflect-delete-send")()

SURFACE *surface

This is the default action for the [refl: delete last] button in project *Surface*. On the first call after a reflect operation all objects created by this operation are deleted, and on every following call only the last object is deleted (example: starting with one object after three reflections there are eight objects, the first call of this method deletes objects 5–8 the next call object 4 and so on).

GRAPE(surface, "refine-send")()

GRAPE(surface, "coarsen-send")()

SURFACE *surface

Send "refine"/"coarsen" to surface ->model to refine/coarsen the underlying geometry (see Timestep and Adapt2d). This is the default action for the [ref-step]/[coa-step] button in project *Refine*.

GRAPE(surface, "refine-all-send")()

SURFACE *surface

Refine (locally) all timesteps in surface ->model using the same refine–field which is taken from the surface ->inter geometry and copied to all steps. Pre– and post–objects are handled automatically but the geometries are not checked for different connectivities. Reflected instances are also handled. This is the default action for the [refine-vid] button in project *Refine*.

GRAPE(surface, "update")()

SURFACE *surface

Update time, frame number, task and for an instance of class Surface by calling "set-frame" and "set-task".

(double *) GRAPE(surface, "get-asso")()

SURFACE *surface

"get-asso" returns the pointer to the (static) ASSO variable which is used for Surface instances (surface ->asso) and for the standard associate ruler rulasso.

Chapter 7

Projects

This section contain 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.

As mentioned before projects can depend on each other, they form their own hierarchy which is shown in figure 7. 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 on.

Some projects just combine other projects to a larger package or provide an interface to methods from the GRAPE library. In the cases you will find references to the appropriate parts of the documentation.

7.1 Projects for Continuum-Mechanics

7.1.1 Project Move-Trace

After adding the project *move-trace* several buttons and rulers to handle the moving and tracing are provided in the first optionmenu *opt0*.
To start one of the the moving methods *first-moving, continue-moving* or *point-moving* and *streak-moving* on the current scene (with a vector valued Fe2d or Fe3d object) press the buttons *first move*, *cont move*, *point move* or *streak move*. A detailed description of these methods can be found in chapter 5.5.2. To use *first-moving* it is neccesary that a test set (a surface (*Triang2d*) or curve (*Triang1d*)) is located at the position of the *next_scene*. With the method *mark-polygon-send* invoked by pressing the corresponding button *mark poly* a polygon can be generated by picking the points on a clip plane. Another possibility to get an object to be moved is to read it from disk using *read gr* in the manager menu *mngr*. The moving methods, parameters can be adjusted by

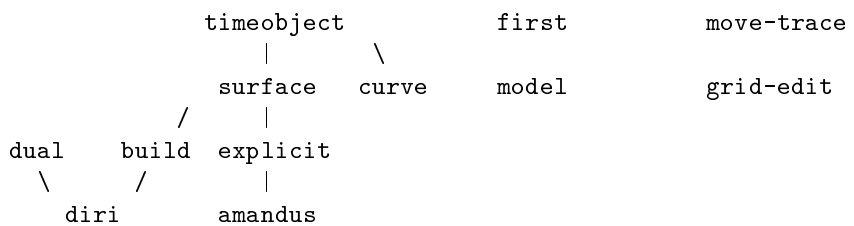


Figure 7.1: Project Dependencies

using the ruler *start-time, end-time, runge-delta, refine-criterium* 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 *first-moving*, *point-moving* and *streak moving*

- as *start* and *end time* of the two tracing methods *tracing-disp* 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 5.5.2, too. The switch buttons alter the display- rather the dynamic-display-method of the moving objects. The *switch display* button shifts the display method of all timescenes stringed together between display and NIL round. The *switch dynamic* button shifts the dynamic display method of all timescenes stringed together between NIL,display and *tracing-disp* round. The method *tracing-disp* 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 *st.suprop1* and *st.suprop2* 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 5.5.3) as a surface consisting of ribbons with these surfaces properties. The width of these ribbons can be modified by varying the ruler *vertical texture*.

7.1.2 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 **Project: add** in the manager of GRAPE. The buttons and rulers of these projects are inserted in the options menu **opt1**.

7.1.2.1 The Project tr2d–edit

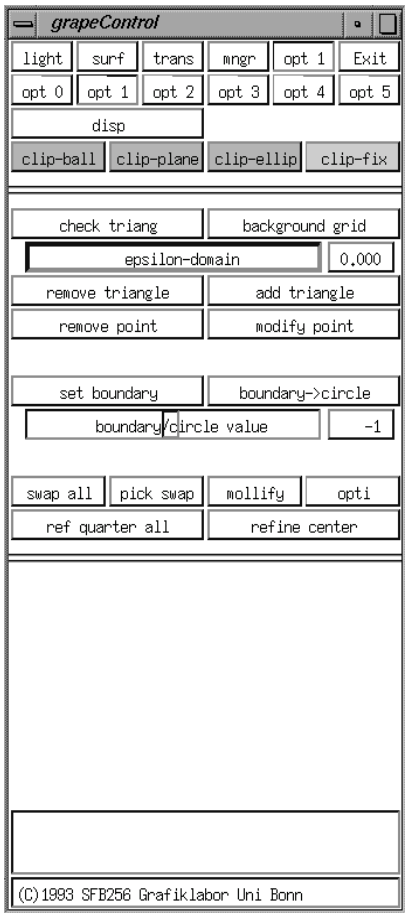


Fig. 1: The layout of the project tr2d–edit

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 features are:

- adding new triangles
- removing triangles
- changing the position of the gridpoints
- removing gridpoints.

Pressing the button **remove 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 **background grid** by pressing the corresponding button. With this tool the user can select background gridpoints. If no background grid is chooser we can select existing mesh points in an user defined epsilon circle. The radius epsilon can be varied with the **epsilon–domain** slider. After pressing the button **background grid** we can choose with the middle mouse button several grid modes which differ in the fineness of the background grid. With the left mouse button we can select a node of the current triangulation and align it to the background grid.
After modification of the triangulation it is recommended to verify its consistency. This can be done by using **check triang**. 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**:

- 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 **ref quarter all**. The local refine technique which divides a triangle into three triangles can be chooser 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 **remove 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. 2, left) and make the grid more regular by swapping the edges (cf. Fig. 2, middle). Then we remove some points in the refined domain and get a mesh similar to the uniform initial grid (cf. Fig. 2, right).

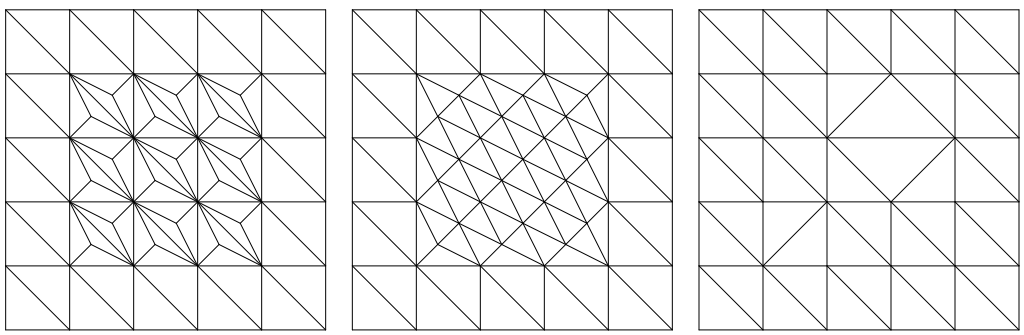


Fig. 2: Uniform mesh after refining with regard to the center in the middle of the domain, swapping and point oriented coarsening

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 bound**. 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 bound** 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 **clip–ball**, **clip–plane** or **clip–ellip**. 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 **clip–fix** 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. 3, middle). Then we have to give the new boundary triangles an unused boundary value. For this we can use the button **set boundary** (see section 7.1.2.1). 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. 3, 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.

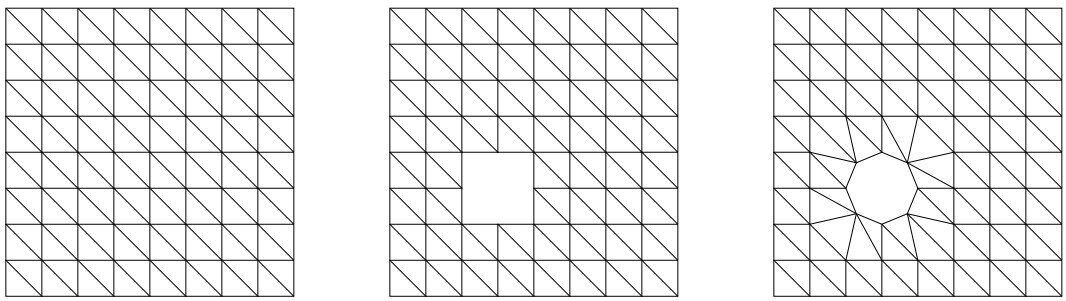


Fig. 3: Another technique to cut a circle out of the mesh

It is possible to refine the mesh or only some triangles and to press the button **boundary->circle** again (cf. Fig. 4).

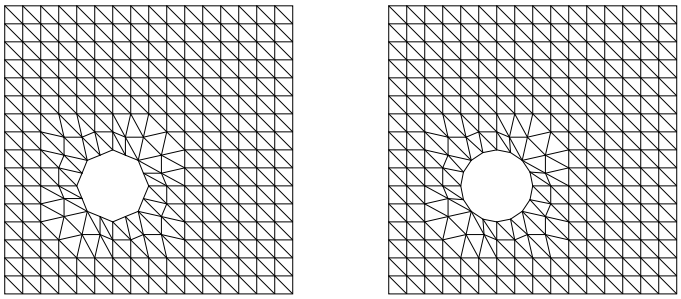


Fig. 4: After refinement the boundary points can moved onto the circle again

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 **swap all**. 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 **pick 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 **opti**.

7.1.2.2 The Project grid–edit

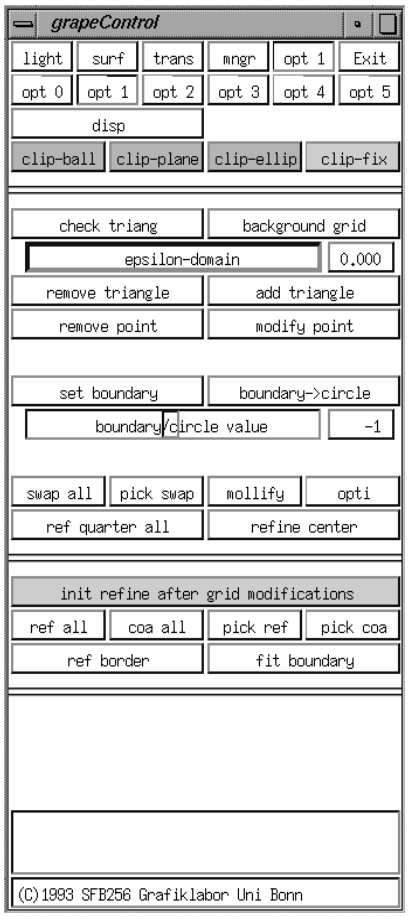


Fig. 5: The layout of the project grid–edit

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

For meshes whose boundaries can be described in an implicit form their shape is stored in the structure

```
typedef struct bnd_function {
        float (*g)();
        void *par;
} BND_FUNCTION; .
```

Where $g(x,y,z,bnd_function \rightarrow par) = 0.0$ 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 {float cx,cy,cz,r;} BALL_PARM_2;
```

```
static float ball(x,y,z,var)
     float x,y,z;
     BALL_PARM_2 *var;
{ float ret;
    ret = (x-var\verb1->1cx)*(x-var\verb1->1cx)
        + (y-var\verb1->1cy)*(y-var\verb1->1cy)
        + (z-var\verb1->1cz)*(z-var\verb1->1cz) - (var\verb1->1r*var\verb1->1r);
    if (var\verb1->1r < 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\verb1->1cx,var\verb1->1cy,var\verb1->1cz)` and radius `var\verb1->1r`. 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 **ref all/coa all** refine/coarsen the whole mesh. After pressing the button **ref all** all triangles of the mesh are divided in half with regard to their refinement edges. With **coa 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 **pick ref** and **pick coa** 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 **pick ref** and picking all triangles along the boundary, the button **ref border** can be activated. It works similarly to **set bound**. 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 after grid modifications** 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 after grid modifications** is pressed before we use any refine/coarsening method on an ADAPT2D mesh. The reason is obvious: the REFINE2D structure in ADAPT2D 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 after grid modifications** but remember after pressing it, it is no longer possible to coarsen any refined triangle.

7.1.2.3 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. 6). 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. 6) with **ref quarter all** (one of the three possibilities to refine a grid) we want to clip out the first ellipse.

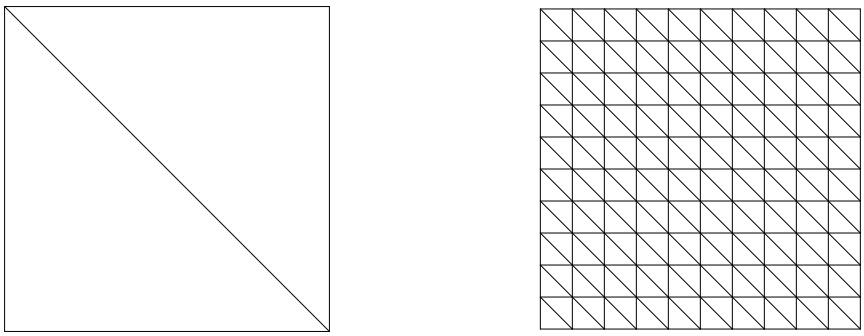


Fig. 6: The starting triangulation and the refined one

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 **clip–ellip**, 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 **clip–fix**. 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 7.

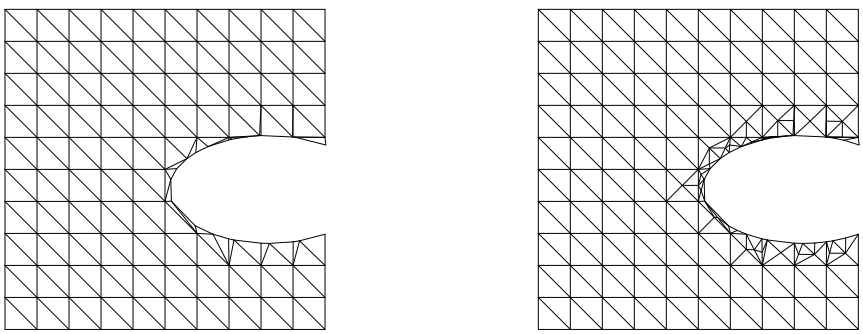


Fig. 7: After clipping out an ellipse and boundary refinement

Now we refine the boundary along the ellipse to get a smoother curve. At first we have to press the button **init refine after grid modifications** 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. 7) 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 **opti** several times (cf. Fig. 8).

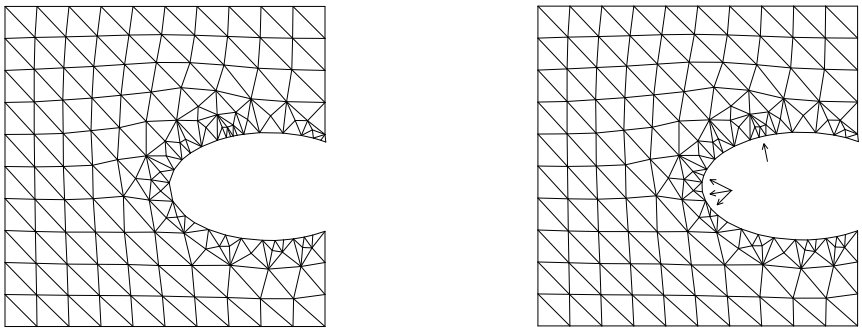


Fig. 8: With mesh optimization and some handwork

In the areas where we are not content with the triangulation we refine or coarse some triangles by hand using **pick ref** or **pick coa** (cf. Fig. 8). Now we apply the same procedure to a second ellipse (cf. Fig. 9), adjusting the parameters and fixing the geometry as described above.

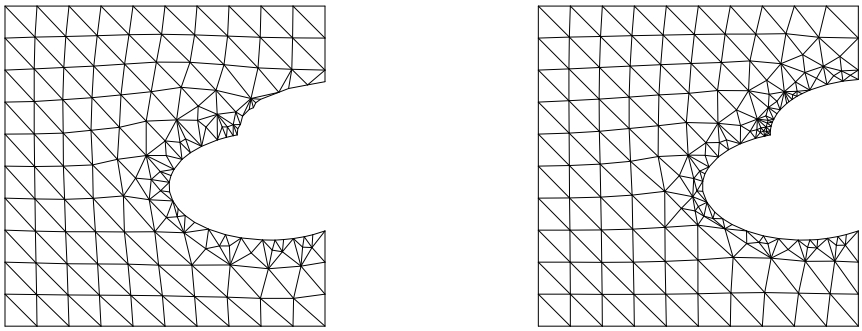


Fig. 9: The same procedure with a second ellipse

Next we try to make the actual triangulation (cf. Fig. 10, left) more regular and uniform along the boundary. To do this we at first use **swap all** and then remove the points which are not needed by picking them after pressing **remove point**. The result shown on in the right hand side of Figure 10 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.

7.1.3 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 *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

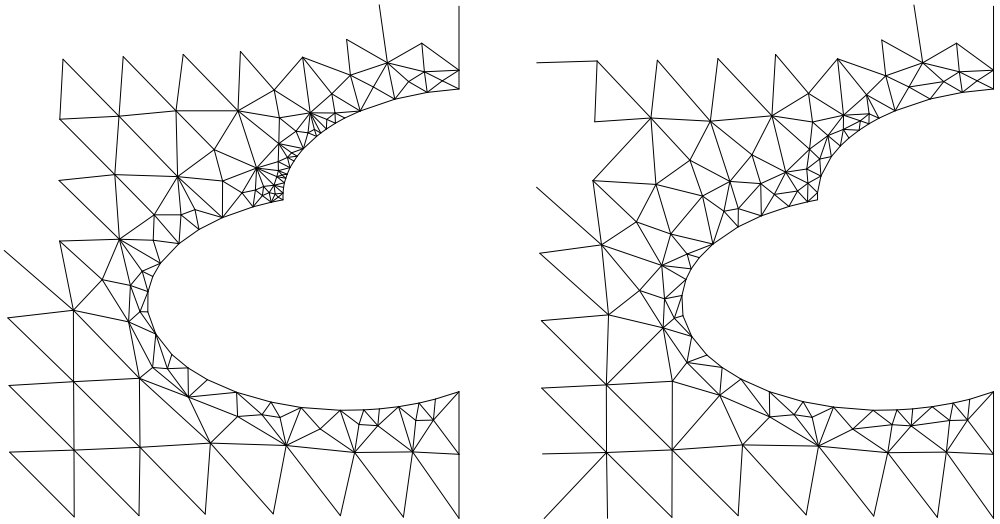


Fig. 10: A zoom to show the 'quality' of the grid along the boundary before and after using **remove point** *and* **swap all**

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.

7.2 Projects for Differential Geometry

7.2.1 Project Time-Object

This projects loads the environment to work at a low level with time dependent objects. It adds a new branch into the scene hierarchy with an instance of class Time_Object. By using description files with extension ***.to** the user can create new objects and manipulate them. For new objects the user only has to supply the data from somewhere, e.g. compute them or read them from a data file.

See section 6.3.1 for the documentation of the Time_Object class and its methods.

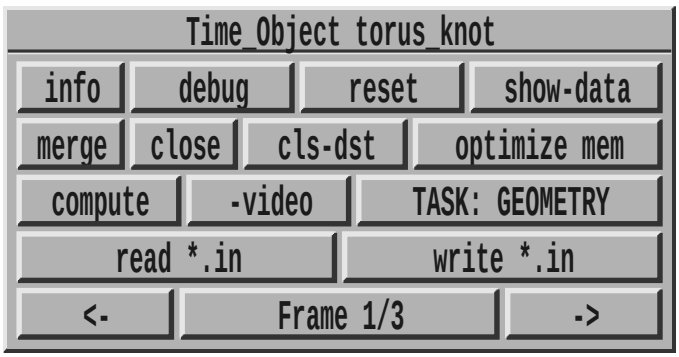


Figure 7.2: User Interface for Project Time_Object

This project adds a user-interface in option menu 0 when it is added. 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.

`info` — Show information in the control window about the object at the current scene node. Pressing the middle mousebutton will return to normal display.

`debug` — Change display to debug mode. In this mode one may pick points or objects in the graphic window and obtain information about the geometry in the window GRAPE was started in (standard output).

`reset` — Initialize the data so that recomputation is possible. The data is not automatically reinitialized when `compute` is pressed again since one may compute data at first at special frames and later compute all other frames by pressing `-video` without recomputing things.

`show data` — This is another debug facility which prints the real geometric data into the unix shell window (standard output).

`merge` — The geometry may consist of several separate geometries, e.g. a cube may be build up of six planes or after having applied reflection operations more geometries come into being. For some applications they need to be merged into one geometry. This affects the scene hierarchy.

`close` — After having applied the merge operation it is sometimes necessary to close a surface along interior boundary lines of the formerly separated objects. This smoothes the shaded picture. The neighbourhood connectivity is extended over whole surface.

Applying this to curves may close start and end point if the are close together.

`cls-dst` — This is a display method therefore you have to press `disp` and `cls-dst`. A new ruler will appear with a value specifying the maximal distance up to which `close` will identify objects.

`optimize mem` — Apply several clean up routine for the geometries, e.g. remove unused memory of the structures, remove unused points, etc.

`compute`, `-video` — The compute routine is send to the actual object computing the actual frame or the whole sequence. Objects computed once may not be recomputed until the `reset` button is pressed.

`TASK:...` — This makes more sense when working with more complicated projects. They allow to compute not only the standard geometry but for example 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. This is used to attack mathematical problems in a step by step approach with first checking simple things and then the more complicated ones. Sub–projects of *Time_Object* should include further tasks.

`read *.in`, `write *.in` — Read or write (not yet implemented) a description file. The type depends on the object currently active in the scene hierarchy. A Time-Object offers and uses description files with extension `*.to` (for a description of the file format see below), other classes use

different file extensions. After clicking on this button a filerequester appears and you can select a file.

There is one thing that should be mentioned. If a time dependent double parameter is defined inside the description file `name.extension` then the system searches in the current directory for a spline description file with name `name-parm-i.sp` where `i` is a number. If this file exists it is used otherwise such a spline is created internally by the system. For further information see description about the class TimeParm (6.3.1.2).

`open *.vd`, `soft still`, `soft video` Functionality is the same as with the `read *.in` button. It is possible to open a description file for writing SOFTIMAGE files for videos. With `soft still` the whole scenery is written as a hierarchical model in a file using SOFTIMAGE format. To write a video sequence one has to open a SOFTIMAGE description file describing various parameters like frame number etc.

These buttons are only available if the SOFTIMAGE driver is part of the library.

`<-`, `Frame i/n`, `->` Walk step by step through the keyframes or select a frame directly by pressing the `Frame i/n` button and typing in the frame number.

Additionally to these button a `geometry displays` button will be added to the option menu. When it is pressed a layer appears which offers some display methods. This will be explained in the next section when more than one geometry is available on class Surface.

In options menu `opt1` appear several rulers depending on the currently loaded object.

With the time ruler the current time at which things happen can be modified. Changing the frame will affect the ruler. The ruler manages the global time for the current object but be aware that other objects may have their own time.

The other rulers in this menu are included by the objects. They are mainly used to modify different parameters of the object. All double parameters are time dependent — that means they are a function over the time. Modifying the value at the ruler will only affect the variables value at this specific time but not change the function itself. After changing the time the ruler is updated again with its function value at the new time. Because of this rulers with changing values slide around when one moves the time ruler. To edit the corresponding function press the button `E` on a ruler. For further information and usage see 8.1.3.9.

The format of the Time-Object input files is very simple, it just contains frame and parameter information:

```
#
# Description file for class Time_Object
#

# task
GEOMETRY;

# frame data
0; 5;           # current frame and number of frames

# parameter data
1; 2;           # number of integer and double parameters

# integer parameters
3;              # n
```

```
# double parameters
2.5;            # radius
1.0;            # perimeter
```

7.2.2 Project Curve

The project curve is the first example of a sub–project of the project *Time-Object*. The only difference compared to its super–project is that a set of additional methods will be made available which always assume that curves are the time dependent geometries one currently works with. So there are methods added for reading description files, allocating storage automatically and compute routines to interpret the description files.

The class Time-Curve is used to compute explicitly given curves. As a sub class of Time_Object it has inherited all its feature, and has additionaly three pointers to coordinate functions. Further it has an additional set of parameters describing the time dependent definition interval and a spline pointer to manage the distribution of the points on the curve. This structure is controlled by description files with extension '.cv'.

The coordinate functions may be typed into the description file. The integer and double parameters may be used as variables in the coordinate functions. The names of the integer parameters have to be

i, j, k, l, m, n, i0, i1, i2, ...

and the names of the double parameter are

a, b, c, d, e, f, g, h, a0, a1, ...

One only has to keep care that the number of parameters is the same as the number of used variables and that the first double parameter corresponds to the variable a and so on.

Remember that the double variables are time dependent so a function ruler will be installed for each variable in the option menu [opt1], allowing modification of the variables over the time.

The following example file will compute a helix with a variable radius. The upper part is identical to a Time-Object file, for Time-Curve the number of poitsn, the parameter (x) interval and the coordinate functions have been added.

```
#
# Description file for class Time-Curve
#

# task
GEOMETRY;

# frame data
0; 5;           # current frame / number of frames

# parameter data
0; 1;           # number of integer and double parameters

# integer parameters

# double parameters
.5;             # radius

# curve parameters
50;             # number of points
0.; 2.*6.283185 # definition interval

# coordinate functions
a * cos(x);     # a corresponds to the first double parameter
a * sin(x);
x / 10.;
```

7.2.2.1 The Time-Curve Structure

```
typedef struct time_curve {
    /* superclass structure (TIME_OBJECT) */
    ...
    /* CURVE */
    CURVE_PARM *curve_parm;          /* Curve parameters (see below) */
    struct {
        t_fun *x, *y, *z;            /* Pointer to coordinate functions */
    } coord-f;
} TIME_CURVE;
```

7.2.2.2 Curve-Parm

Curve_Parm is the structure controlling the special curve parameters.

```
typedef struct curve_parm {
    /* superclass structure (INSTANCE) */
    ...
    /* PRIVATE */
    FUNCTION_RULER *min_rul;
    FUNCTION_RULER *max_rul;
    FUNCTION_RULER *dstrbn_rul;
    t_fun *t_min, *t_max;
    /* USER */
    int num_of_points;             /* number of points */
    SPLINE1D *dstrbn;              /* spline for distribution of points */
    double min, max;               /* definition interval */
} CURVE_PARM;
```

GRAPE(curve_parm, "delete")()

CURVE_PARM *cur_parm

Delete a Curve_Parm instance. All data is deleted using "list-free", all curve_parm rulers are removed from the manager and deleted.

GRAPE(curve_parm, "list-alloc")()

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

CURVE_PARM *curve_parm

At the moment all memory allocation for a Curve-Parm instance is done when the input-file ('*.cv') is read in, therefore "list-alloc" is an empty method which exists only for consistency.

"list-free" deletes all data including the function rulers and the spline which have been allocated in "read-data".

7.2.2.3 Memory Methods on Time-Curve

(TIME-CURVE *)GRAPE(Time_Curve, "new-instance")(name)

CLASS *Time_Curve
char *name

Create a Time_Curve instance. First "new-instance" on the superclass Time_Object is called and then the Curve_Parm instance time_curve ->curve_parm is created.

GRAPE(time_curve, "delete")()

TIME_CURVE *time_curve

Delete a Time_Curve instance. All memory allocated by "list-alloc" and "read-data" is freed using "list-free", then the instance itself is deleted.

GRAPE(time_curve, "list-alloc")()

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

TIME_CURVE *time_curve

"list-alloc" first calls "list-alloc" on the superclass Time_Object and then allocates time_curve ->geometry using "alloc" and "create-objects" on TimeStep. Finally for each timestep a Triang1d instance with time_parm ->num_of_points is allocated.

"list-free" deletes all data which has not been allocated by "new-Instance". This is done by calling "list-free" on the superclass and on time_curve ->time_parm.

7.2.2.4 Compute Methods on Time-Curve

GRAPE(time_curve, "compute-send")()

TIME_CURVE *time_curve

Compute the points of time_curve using the time_curve ->curve_parm parameters.

7.2.2.5 I/O Methods on Time-Curve

(LIST *)GRAPE(time_curve, "open-in")(list, device, picked_name)

TIME_CURVE *time_curve
LIST *list
TEXTDEVICE **device
char *picked_name

Create the list of input files for class Time_Curve ('*.cv'). This method is called when the [read *.in] button in the options menu is pressed.

GRAPE(time_curve, "read-data")(name)

TIME_CURVE *time_curve
char *name

Read the input file 'name.cv' and create the parameter lists and the associated rulers for time_curve ->time_parm and the rulers associated to time_curve ->curve_parm.

7.2.2.6 Other Methods on Time-Curve

GRAPE(time_curve, "get-object")(inst, time)

TIME_CURVE *time_curve
INSTANCE *inst
double time

Create the time_curve ->inter instance for time time (by calling "get-object" on time_curve ->geometry) and assign it to inst. See 3.5.1.

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

TIME_CURVE *time_curve
LIST *list

Create info list entries for a Time_Curve instance. Entries are class, name, refcount, current frame number, number of frames, time and number of points.

GRAPE(time_curve, "make-tree-send")()

TIME_CURVE *time_curve

Create a time_curve ->tree. This method adds a new TimeScene node to the hierarchy tree (follow environment for class Time_Curve) which has the object time_curve ->geometry. 6.3.2, below for a further explanation.

7.2.3 Project Surface

This sub–project of *Time-Object* supplies the environment to work with instances of the class Surface (the class Surface and its methods were already introduced in section 6.3.2). This class is designed to give a basis to work with time dependent surfaces. The surfaces may have boundaries, or parameter lines may be available, they are stored as Geom2d (see 5.3.4) instances.

Additional to the "main" geometry further data sets are available: the stereographic projection of the normals onto the plane, i.e. the gauss map, and a second data set corresponding with the same connectivity as the geometry. This may be used for different purposes: for example handling the data of a two parameter family (additional to the time parameter) in memory.

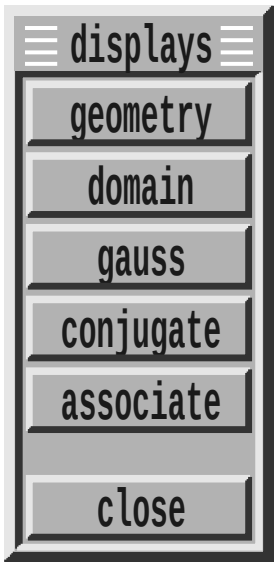


Figure 7.3: Display Layer for Project Surface

Clicking on the `geometry displays` button will pop up a layer that allows to select a display method that displays a specific data set (domain, gauss, etc.). This is useful when geometry you are currently working on (selected by the task button) differs from the one that should be displayed.

`associate` invokes an interpolation routine (controlled by a ruler) between the geometry and the conjugate data set. So together with the time ruler a two parameter family of surfaces may be handled at the same time.

The Surface project adds a new panel to the Time-Object interface, the top three button deal with reflection, the other checkboxes determine which parts of the Geom2d instances are visible.

A few words about reflection of time dependent objects. When reflecting static objects we pick three points on the object to specify a symmetry plane, two points for a straight line or one point for a point reflection. Then the operation is well defined and can be executed.

Figure 7.4: User Interface for Project Surface

For time dependent objects this is quite different. During picking we see on the screen 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 just visible symmetry plane. So we cannot just pick special points or specify a symmetry plane, we can only pick "meta structures" such as for example boundary curves.

Picking a symmetry line therefore is done by picking a boundary curve. The object itself has to know what this boundary curve is at each time step and to reflect respectively. With this method the picking is independent of the connectivity and the special position of the symmetry line.

`reflect`

Pressing this buttons changes the mode to reflection. The current object will be drawn in green (in grid mode). Picking of a boundary curve specifies the symmetry line. Usually the system will automatically detect the type of symmetry, e.g. reflection at a plane, at a line, inversion at a circle etc. For symmetry operations as inversion at a point two boundary curves must be choosen, their intersection point determines the center of reflection. The reflection type may be set to a specific reflection operation using the button `refl: config?` therefore disabling the automatic choice of type. Choose a symmetry line by picking a point belonging to a curve. Invoke the reflection, be guided by the messages at the bottom of the control window. Usually the left mouse button picks, right mouse button accepts choices or executes, and the middle mouse button exits without changes. The newly created objects are added to the scene hierarchy. To undo things these objects may be deleted from the scene hierarchy inside the manager menu `mngr`. Be careful if you are not in standard display mode but for example in domain display mode (see below). When you reflect you will normally see the object belonging to the current `TASK:@dots` button.

`refl: del last`

This button allows you to undo the last reflection. Further button presses remove previous generated objects object by object until only one object is left.

`refl: config`

Pressing this button will pop up a layer will allows to select which reflection type should be used (left half) and what objects should be reflected(right half).

Only one reflection type can be active. Without making an explicit choice the default is "automatic detection". Reflection is possible at a plane, 180 degree rotation at a straight line, inversion at a point or 180 degree rotation around the normal of a point or inversion at a circle. Also reflection operations for hyperbolic and spherical geometries a possible.

The checkbox `all` lets you apply the reflection operation to all visible objects (selected) or to select specific object to be reflect with the mouse (not selected) when you have pressed `reflect`. The

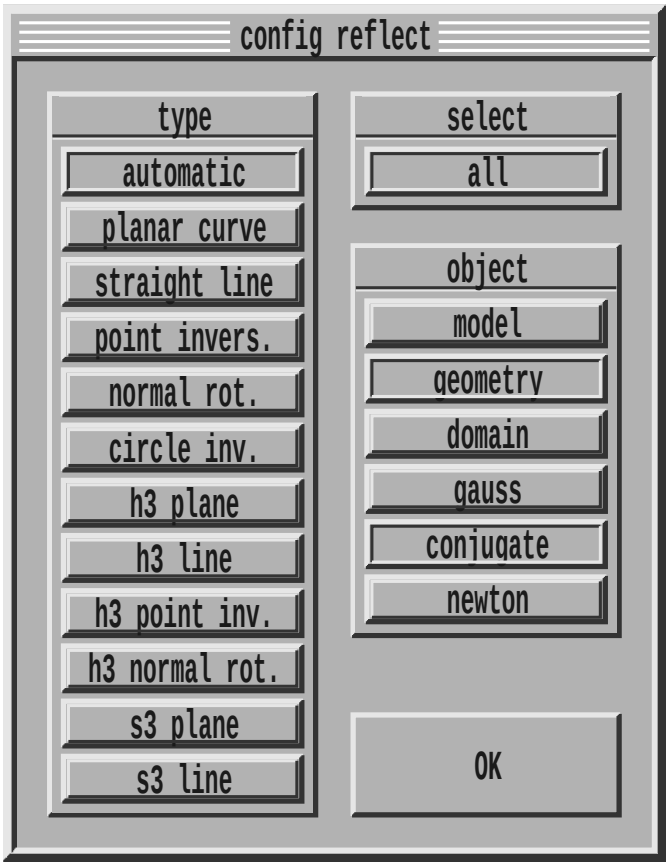


Figure 7.5: Reflection Config-Layer in Project Surface

other switches allow to select the objects that should be reflected (in the example the geometry and the conjugate). If you select `model` the object you have set using the `TASK:...` button is used.

`geom` This checkbox selects whether the Geom2d surface should be shown. This and the following checkboxes depend on the task, therefore it is possible to select different parts of the Geom2d instance to be displayed for each task.

`bnd` Selects if the boundary should be drawn. Some rulers will be added if this checkbox is selected, they allow to control the discretization and the radius of the boundary tube.

`hor` `ver` Select if the (horizontal/vertical) parameter lines should be displayed. The parameter lines might not be available, for example when they were not computed and the surface was closed.

7.2.4 Project Surface-Build

This sub–project of *Surface* is used as a surface generator to create topological surfaces of arbitrary type and genus. It is possible to build a surface by pasting a collection of triangular or quadrilateral patches or patches bounded by a single polygon together. The description of the patches has to be put in a file with the extension *.bd, it consists of a set of vertices and a list of patches that are bounded by curves bounded by the given vertices.

A speciality is that a flag for a vertex component can be set such that this component can later be modified by a ruler or modified as a function over the time. Marking different components with the same flag allows it to move different points simultaneously.

This class has one additional instance variable of class Build-Parm which is used to store information about the different kinds of patches, their number etc.

To create surfaces of this class just add the project *Build* with the `Project: add` button and read a description file of type '*.bd'.

The description file this class is quite complex, you should take a look at some of the files in the demo directory of the GRAPE distribution.

```
#
# Description file for class Surface-Build
#

# task
GEOMETRY;

# frame data
0; 5;           # current frame / number of frames

# parameter data
0; 0;           # number of integer and double parameters

# integer parameters
# double parameters

# part/patch data
3;              # number of patches, i.e. parts
8;              # number of vertices

# vertex list
# (ix, iy, iz) (x, y, z) flags for each coord, coordinate vector of vertex
# the integers are flags: if a flag is !=0 this means
# that this component shall be time dependent, therefore
# a function ruler for this variable will be installed.
# if different the flags for different components are identical
# then these variable will be identified and controlled by the
# same function ruler. With this facility one can e.g. modify
# boundary lines or planes over the time
0; 0; 0; 0.; 0.; 0.;
0; 0; 0; 0.; 1.; 0.;
0; 0; 0; 0.; 2.; 0.;
0; 0; 0; 0.; 0.; 1.;
0; 0; 1; 0.; 1.; 1.; # the z-component of this vertex may be animated
0; 0; 0; 0.; 2.; 1.;
2; 0; 0; 1.; 1.; 0.; # both x-components may be animated at the same time
2; 0; 0; 1.; 1.; 1.;

# patch list
# flag, bnd-type, points-per-patch, {p0, @dots{}, pn}, ulines, vlines, usub, vsub
0; 0; 4;  0; 1; 4; 3;   4; 4; 0; 0;
0; 0; 4;  6; 1; 4; 7;   4; 4; 0; 0;
0; 0; 4;  2; 1; 4; 5;   4; 4; 0; 0;
```

The patch list items have the following meanings

flag: used to change orientation of a patch (1 = change orientation) bnd-type: 0 = no further information, 1 = further edge information (see below) points-per-patch: number of points for this patch p0, ..., pn: indices of the vertices above which build up this patch ulines: number of lines between first and

second vertex vlines: number of lines between second and third vertex usub, vsub: number of subdivisions between two lines along u– or v–direction

In case of bnd-type == 1 the user has to supply information for each edge of the form:

octal; 0; [int; int;]

where the two ints are optional but required for special octals. They refer to two vertices of the coordinate list, but these coordinates will be used differently, see below. The octal bits have the following meaning:

The lower four bits restrict the curve to special geometries, e.g. all boundary points are restricted to the specified plane. Such information is used e.g. in minimization algorithms to determine free boundaries.

```
00010 = curved     fully restricted, i.e. fixed
00020 = straight   restricted to straight line
00040 = planar     restricted to plane
00100 = thread     arbitrary but fixed length
```

The next four bits types determine the initial shape of the boundary, e.g. the boundary is part of a circle. In all cases the curve goes through the endpoints of the edge as in the default case, so the two additional indices refer to two vertices of the vertex list which contain additional information as e.g. midpoint and normal of the circle. Behind the type are the optional/required indices:

```
00200 = straight
00400 = circle     midpoint, normal to circle plane
01000 = helix      axis start point, axis end point
02000 = spline     tangent vectors at end points
```

7.2.4.1 The Surface-Build Structure

```
typedef struct surface_build {
    /* superclass structure (SURFACE) */
    ...
    /* Surface_Build */
    BUILD-PARM *build_parm;
} SURFACE_BUILD;
```

7.2.4.2 Build-Parm

```
typedef struct build_parm {
    /* superclass structure (INSTANCE) */
    ...
    /* USER */
    int nop;
    double **vec;
    t_fun **tvec;
    int *funrul_flag;
    FUNCTION_RULER **funrul;
    int num_parts;
    int *flag;
    int *bnd_type;
    int *type;
    int *nov;
    int **vertex;
    int *num_u_lines, *num_v_lines;
    int *num_u_sub, *num_v_sub;
    int **bnd_spec;
} BUILD_PARM;
```

GRAPE(build_parm, "delete")()

BUILD_PARM *build_parm

Delete a Build_Parm instance. All data is deleted using "list-free", all build_parm rulers are removed from the manager and deleted.

GRAPE(build_parm, "list-alloc")()

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

BUILD_PARM *build_parm

"list-alloc" allocates the memory for all build-parm variables (flags and variables for rulers, boundaries, lines etc). Also the memory for the function_ruler arrays is allocated, the rulers themselves are allocated while reading the input file (in method "read-data").

"list-free" deletes all data including the function rulers (not only the pointers allocated in "list-alloc") which are removed from the manager and deleted.

GRAPE(build_parm, "set")(on_off)

BUILD_PARM *build_parm
int on_off

Activate (on_off == ON) or deactivate (on_off == OFF) all rulers belonging to build_parm.

GRAPE(build_parm, "switch-send")()

BUILD_PARM *build_parm

Toggle the status of all rulers belonging to build-parm (ON <-> OFF).

7.2.4.3 Memory Methods on Surface-Build

(SURFACE_BUILD *)GRAPE(Surface_Build, "new-instance")(name)

CLASS *Surface_Build
char *name

Create an Surface_Build instance with name name. First "new-instance" on the superclass Surface is called and then the Build_Parm instance surface_build ->build_parm is created.

GRAPE(surface_build, "delete")()

SURFACE_BUILD *surface_build

Delete an Surface_Build instance. All memory allocated by "list-alloc" and "read-data" is freed using "list-free", then the instance itself is deleted.

GRAPE(surface_build, "list-alloc")()

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

SURFACE_BUILD *surface_build

"list-alloc" allocates the memory needed for the Surface-build instance. In contrast to "list-alloc" on the classes Explicit and Amandus each part is stored in a separate Geom2d instance.

"list-free" deletes all data which has been allocated by "list-alloc" and "read-data". The functions rulers associated with surface_build are removed from the manager and deleted.

GRAPE(surface_build, "merge-send")()

SURFACE_BUILD *surface_build

The geometries are merged (into only one Geom2d instance) by sending "merge-send" to the superclass Surface. All (time dependent) rulers associated with surface-build are deactivated (by calling "set"(OFF)) and the flag for all parts is set to MERGED to avoid recomputation.

7.2.4.4 Display Methods on Surface-Build

GRAPE(surface_build, "ref-curv-disp")()

SURFACE_BUILD *surface_build

Interactive method for selecting edges for refinement. This method is only needed to allow the display of parameter lines and/or the boundaries. See ->"ref-curv-disp" on Adapt2d for further explanation.

7.2.4.5 Compute Methods on Surface-Build

GRAPE(surface_build, "compute-video-send")()

SURFACE_BUILD *surface_build

Compute all frames for surface_build by sending "compute-send" to each frame. The current frame number is saved and restored after the computation has been done.

GRAPE(surface_build, "compute-send")()

SURFACE_BUILD *surface_build

Compute the surface using the surface_build ->build_parm data. Several "build_" methods on ->Geom2d are used to create different kinds of patches.

7.2.4.6 I/O Methods on Surface-Build

(LIST *)GRAPE(surface_build, "open-in")(list, device, picked_name)

SURFACE-BUILD *surface_build
LIST *list
TEXTDEVICE **device
char *picked_name

Create the list of input files for class Surface_Build ('*.bd'). This method is called when the [read *.in] button in the option menu is pressed.

GRAPE(surface_build, "read-data")(name)

SURFACE_BUILD *sb
char *name

Read the input file name.bd and create the parameter lists and the rulers associated with surface_build. After reading the input file "compute-video-send" is used to compute all frames.

7.2.4.7 Other Methods on Surface-Build

GRAPE(surface_build, "assign-cons-send")()

SURFACE_BUILD *surface_build

Compute the parameter lines for surface_build. This method calls "assign-cons" with the necessary parameters. Computation is only possible if the parts have not been merged!

GRAPE(surface_build, "assign-cons")(parts, u_lines, v_lines, u_subs, v_subs)

SURFACE-BUILD *surface_build
int parts
int *u_lines, *v_lines
int *u_subs, *v_subs

Extract parameter lines of an Surface_Build instance and store them is the Cons instances hor and ver of the underlying Geom2d instance.

GRAPE(surface_build, "switch-comp-send")()

SURFACE_BUILD *surface_build

Toggle the status (ON/OFF) of all rulers belonging to surface_build by sending "switch-send" to surface_build ->build_parm and calling "switch-comp-send" on the superclass Surface.

7.2.5 Project Explicit

This class, a subclass of Surface, is designed to work with explicitly given surfaces, i.e. parametrized surfaces. Additional to this class are instance variables for the three coordinate functions and domain-parm with additional parameters to control the parametrization of the domain.

The user only has to supply a description file of type `*.ex` with the coordinate functions and information about the domain and can then compute the surface with the standard procedure by loading the project and reading the description file.

The coordinate functions may be typed into the description file. The integer and double parameters may be used as variables in the coordinate functions. The names of the integer parameters have to be

i, j, k, l, m, n, i0, i1, i2, ...

and the names of the double parameter are

a, b, c, d, e, f, g, h, a0, a1, ...

One only has to keep care that the number of parameters is the same as the number of used variables and that the first double parameter corresponds to the variable a and so on.

Remember that the double variables are time dependent so a function ruler will be installed for each variable in one of the option menus allowing modification of the variables over the time.

Modifying the rulers will directly lead to a recomputation of the data. This behaviour is different from other structures, where computation needs more time and therefore is only done after `reset` and `compute` or `compute-video` have been pressed.

To experiment with this class copy one of the *.ex files from the GRAPE distribution and modify the values. The format for the first part of the file is identical to class Time-Object. Then the domain parameters and the coordinate function follow.

```
#
# Description file for class Explicit
# (example is a cycloid with variable quotient of
# max to min radius)
#

# task
GEOMETRY;

# frame data
0; 5;           # current frame / number of frames

# parameter data
0; 1;           # number of integer and double parameters

# integer parameters

# double parameters
1.;            # lambda: <1 long, >1 knotted cycloids

# part, number of parts
1; 1;
```

```
# flag, ulines, vlines, usub, vsub
0; 15; 10; 0; 0;

# size of domain: (umin, vmin) - (umax, vmax)
3.1415926; 0.0; 5. * 3.1415926; 2. * 3.1415926;

# coordinate functions
x - a * sin(x);             # a == lambda
cos(y) * (1. - a * cos(x));
sin(y) * (1. - a * cos(x));
```

7.2.5.1 The Explicit Structure

```
typedef struct explicit {
    /* superclass structure (SURFACE) */
    ...
    /* EXPLICIT */
    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;
} EXPLICIT;
```

7.2.5.2 Domain-Parm

```
typedef struct domain_parm {
    /* superclass structure (INSTANCE) */
    ...
    /* PRIVATE */
    DOMAIN_PARM_RUL rul; /* Ruler for all domain parameter */
    /* DOMAIN_PARM */
    int num_parts;       /* Number of parts */
    int max_parts;       /* Maximum number of parts */
    int *flag;           /* Flags for orientation of parts */
    int *num_u_lines;    /* Number of lines for each   */
    int *num_v_lines;    /* part in u- and v-direction */
    int *num_u_sub;      /* Number of subdivisions of u- */
    int *num_v_sub;      /* and v-lines for each part    */
    SPLINE1D **u_dstrbn; /* Line distribution for the  */
    SPLINE1D **v_dstrbn; /* parts (u- and v-direction) */
    t-fun **u_min;
    t-fun **v_min;
    t-fun **u_max;
    t-fun **v_max;
    double **u_min_d;
    double **v_min_d;
    double **u_max_d;
    double **v_max_d;
} DOMAIN_PARM;
```

GRAPE(domain_parm, "delete")()

DOMAIN_PARM *domain_parm

Delete a Domain_Parm instance. All data is deleted using "list-free", all domain-parm rulers are removed from the manager and deleted.

GRAPE(domain_parm, "list-alloc")()

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

DOMAIN_PARM *domain_parm

"list-alloc" allocates the memory for all domain-parm variables: dstrbns, min, max, number of lines etc. Also the memory for the function_ruler arrays is allocated but the rulers have to be allocated separately, this is done in g-domain-get-parm-data (called in method "read-data") while reading the input file.

"list-free" deletes all data including the function rulers (not only the pointers allocated in "list-alloc") which are removed from the manager and deleted.

GRAPE(domain_parm, "set")(on_off)

DOMAIN_PARM *domain_parm
int on_off

Activate (on_off == ON) or deactivate (on_off == OFF) all rulers belonging to domain_parm.

GRAPE(domain_parm, "switch-send")()

DOMAIN_PARM *domain_parm

Toggle the status of all rulers belonging to domain-parm (ON ¡-> OFF).

7.2.5.3 Memory Methods on Explicit

(EXPLICIT *)GRAPE(Explicit, "new-instance")(name)

CLASS *Explicit
char *name

Create an Explicit instance with name name. First "new-instance" on the superclass Surface is called and then the Domain_Parm instance explicit ->domain-parm is created.

GRAPE(explicit, "delete")()

EXPLICIT *explicit

Delete an Explicit instance. All memory allocated by "list-alloc" and "read-data" is freed using "list-free", then the instance itself is deleted.

GRAPE(explicit, "list-alloc")()

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

EXPLICIT *explicit

"list-alloc" allocates the memory needed for the Explicit instance. The number of points needed for each geometry is calculated and then "list-alloc" on the superclass Surface is called to allocate the memory. In contrast to "list-alloc" on Class Surface_Build all parts are stored in one Geom2d instance.

"list-free" deletes all data which has not been allocated by "new-instance" by calling "list-free" on the superclass Surface and then deleting the domain-parm instance.

GRAPE(explicit, "merge-send")()

EXPLICIT *explicit

Send "merge-send" to the superclass Surface to merge all geometries, and deactivate all explicit ->domain_parm rulers (by sending "set"(OFF)).

7.2.5.4 Compute Methods on Explicit

GRAPE(explicit, "compute-send")()

EXPLICIT *explicit

This is the main compute–routine for Explicit instances. Is uses some local (static) routines to check if enough memory is available and to allocate memory if needed (check-geom2d-mem) and to create a simple connectivity (make-geom2d-conn, copy-geom2d-conn). Then the points are computed calling "compute-explicit" and finally the boundaries and parameter lines are updated. Currently only one part is supported and only domain and geometry are computed.

GRAPE(explicit, "compute-explicit")(geom, num, z)

EXPLICIT *explicit
GEOM2D *geom
int num
COMPLEX z

Compute the geom ->x/y/z coordinates with index num over (z.x, z.y) in the domain. The x/y/z-functions can be specified explicitly, this has to be done if the functions depend on parameters (theta-shift, kat-hel, etc). If no special function are given the function from the input–file are used.

GRAPE(explicit, "set-compute-arg")()

EXPLICIT *explicit

To add user defined parameter transformation the user can replace this method with his own "set-compute-arg" method which should initialize explicit ->compute_arg with a user defined get-argument function.

7.2.5.5 I/O Methods on Explicit

(LIST *)GRAPE(explicit, "open-in")(list, device, picked_name)

EXPLICIT *explicit
LIST *list TEXTDEVICE **device
char *picked_name

Create the list of input files for class Explicit ('*.ex'). This method is called when the [read *.in] button in the options menu is pressed.

GRAPE(explicit, "read-data")(name)

EXPLICIT *explicit
char *name

Read the input file name.ex and create the parameter lists and the rulers associated with explicit. After reading the input file "compute-video-send" is used to compute all frames.

7.2.5.6 Other Methods on Explicit

GRAPE(explicit, "assign-cons-send")()

EXPLICIT *explicit

Extract parameter lines of an Explicit instance and store them in the Cons instances hor and ver of the underlying Geom2d instance. This method computes the pipes for connectivities created with "build-quadr-send" on class Geom2d or make-geom2d-conn.

GRAPE(explicit, "switch-comp-send")()

EXPLICIT *explicit

Toggle the status (ON/OFF) of all rulers belonging to explicit by sending "switch-send" to explicit ->domain-parm and calling "switch-comp-send" on the superclass Surface.

7.2.6 Project Amandus

Amandus is a subclass of Explicit designed to work with minimal surfaces defined via the Weierstrass representation formula. It contains further instance variables for the Weierstrass formulas and some other functions. Further a new TimeStep to store the image of an arbitrary function during the computation is supplied. This is mainly used to store the result of a Newton iteration by which the analytic continuation of functions along the integration path is computed. This is usually a very unstable process and needs a lot of attention.

The Weierstrass functions and a function which computes the domain for your surface have to be written in C code and linked to the GRAPE program. These functions are declared to the system using the methods "set-weier-f" and "set-compute-arg". For a simple example on how to add your own surfaces see the files `my-arg-weier.c` and `my-cat-enn.am` in the demo directory of the GRAPE distribution.

You can choose between two different representation formulas.

The local version is

$\int (1-g^2), i*(1+g^2), 2*g*f*dz$

where only the two complex functions g(z) and f(z) have to specified.

The global version is

$\int (g-1/g), i*(g+1/g), 2*dh$

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 too in dzpoly. Supplying the derivative dwpoly of poly with respect to dh is optional but will increase the speed of the Newton method very much.

Inside the description file one has to choose between local and global version by setting the mode to KARL (Weierstrass) or HERMANN (Amandus Schwarz).

The first part of the description file again contains the Time-Object part, that is the time dependent parameters. Then the number of surface patches and the patch data follows, this is basically the description of the domain for the integration.

```
#
# Description file for class Amandus
#

# task
GEOMETRY;

# frame data
0; 5;           # current frame / number of frames

# parameter data
0; 2;           # number of integer and double parameters

# integer parameters

# double parameters
1.;             # rho
0.8556          # r

# part, number of parts
1; 1;
```

```
# part 1
0; 8; 10;       # flag (orientation); number of u- / v-lines
0; 1;           # number of intermediate points u / v
3.; 0.; 0.; 3.141592653;     # (umin, vmin) - (umax, vmax)

# local/global Version
KARL
```

7.2.6.1 The Amandus Structure

```
typedef struct amandus {
    /* superclass structure (EXPLICIT) */
    ...
    /* AMANDUS */
    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;
    NODE *node, *inode;
} AMANDUS;
```

7.2.6.2 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 nodes necessary for the computation are allocated. The memory for the timesteps has to be allocated separately by calling "list-alloc".

GRAPE(amandus, "delete")()

AMANDUS *amandus

Delete an instance of class Amandus. All data is deleted using "list-free".

GRAPE(amandus, "list-alloc")()

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

AMANDUS *amandus

"list-alloc" allocates the memory lists for the Amandus instance. First "list-alloc" on the superclass Explicit is called and then the number of points needed for each geometry is calculated (like in "list-alloc" on Explicit) to allocate the memory for amandus ->newton.

"list-free" deletes all data which has not been allocated by "new-instance" by calling "list-free" on the superclass Explicit and sending "delete" to amandus ->newton.

GRAPE(amandus, "merge-send")()

AMANDUS *amandus

Send "merge-send" to the superclass Explicit and to amandus ->newton to merge all geometries. By calling explicit-"merge-send" all rulers associated with the Amandus instance are deactivated.

7.2.6.3 Display Methods on Amandus

GRAPE(amandus, "newton-disp")()

AMANDUS *Amandus

Display the time section of amandus ->newton for time amandus ->time. The time section is created by calling "get-object"(surface ->time) and stored in amandus ->inter which then is displayed.

7.2.6.4 Compute Methods on Amandus

GRAPE(amandus, "compute-send")()

AMANDUS *amandus

This is the main compute–routine for Amandus instances (weierstrass representations). 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 on exit of computation.

GRAPE(amandus, "compute-node")(old, new)

AMANDUS *amandus
NODE *old, *new

Compute a new point from a given old point. Depending on the task first domain and then newton, gauss and surface/conjugate are computed (see methods below).

GRAPE(amandus, "compute-domain")(new)

AMANDUS *amandus;
NODE *new;

Compute a new point in the parameter domain of a minimal surface. The given point new is only assigned, it has been transformed via the get-argument function in amandus-"compute-send".

GRAPE(amandus, "compute-explicit")(new)

AMANDUS *amandus
NODE *new

Compute a surface whose coordinate functions are given explicitly. Only for historical reasons, not to be used any more.

GRAPE(amandus, "compute-newton")(old, new)

AMANDUS *amandus
NODE *old, *new

Analytic continuation of `w` from old ->z to new ->z by introducing `NUM_OF_STEPS` intermediate points stored in `NEWTON_VEC_Z`. The values of `w` along this path are stored in `NEWTON_VEC_W`. They will be used as initial values to compute `w`. This function gives `w` at `z`, where `z` in interval [old ->z, new ->z] (e.g. during integration): assuming `NEWTON_VEC_Z` and `NEWTON_VEC_W` are computed for this interval by use of `"compute-newton"` call `get-newton` with argument `z`. It looks in

NEWTON_VEC_Z for the right position and uses the corresponding value in NEWTON_VEC_W as initial value for a newton iteration.

get_newton: As described above this routine searches for initial value of w in NEWTON_VEC_W and applies further newton iterations if necessary. The variable ROM_POSITION is supplied by romberg integration and gives the right position of z in NEWTON_VEC_Z.

cnewton: Main newton iteration routine, complex version. If also dzf is supplied by the user, a much better initial value will be computed.

GRAPE(amandus, "compute-gauss")(new)

AMANDUS *amandus
NODE *new

Compute of image of the domain points under gauss mapping (using amandus ->g).

GRAPE(amandus, "compute-surface")(old, new)

AMANDUS *amandus
NODE *old, *new

Compute the points (integration) and the normals (stereographic projection of the gauss map) of the minimal surface and its conjugate surface in three–dimensional space.

GRAPE(amandus, "get-weier-f")()

AMANDUS *amandus

Initialize all function pointers of amandus ->weier-f depending on amandus ->name. This function is called in "read-data" after the input file has been read. To supply his own weierstrass data the user has to call

GRAPE(amandus, "set-weier-f")()

AMANDUS *amandus

To add user defined weierstrass data the user can replace this method with his own "set-weier-f" method which should (depending on amandus ->name) initialize the amandus ->weier-f function pointers.

7.2.6.5 I/O Methods on Amandus

(LIST *)GRAPE(amandus, "open-in")(list, device, picked-name)

AMANDUS *amandus
LIST *list
TEXTDEVICE **device
char *picked-name

Create the list of input files for class Amandus ('*.am'). This method is called when the [read *.in] button in the options menu is pressed.

GRAPE(amandus, "read-data")(name)

AMANDUS *amandus
char *name

Read the input file name.am and create the parameter lists and the rulers associated with amandus.

7.2.6.6 Other Methods on Amandus

GRAPE(amandus, "assign-cons-send")()

AMANDUS *amandus

Extract parameter lines of an Amandus instance and store them in the Cons instances hor and ver of the underlying Geom2d instance. This method computes the pipes for the connectivity created by "compute" on Amandus (which is at the moment different from the connectivity used in Explicit or Surface_Build).

GRAPE(amandus, "close-send")()

AMANDUS *amandus

Send "close-send" to amandus ->newton and to the superclass Explicit to close all geometries.

GRAPE(amandus, "reset-send")()

AMANDUS *amandus

Send "reset" to amandus ->newton and to the superclass Explicit to reset the underlying geometries and the status of all frames.

7.2.7 Project Dual

The project *Dual* contains additional methods concerning the Dirichlet energy minimizer and a conjugater for discrete harmonic maps. The methods can be applied to arbitrary surfaces. Usually initial surfaces are created with project *Explicit* or *Build*. There is also a version of the Dirichlet minimizer for surfaces in hyperbolic space but it is still in an experimental state.

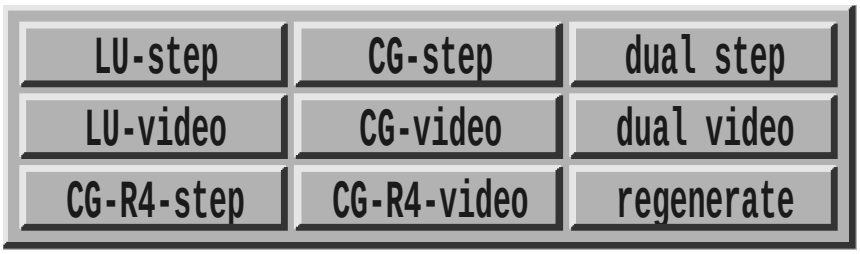


Figure 7.6: User Interface of Project Dual

The suffixes `...-step` and `...-video` indicate that the action is applied to the current frame or to all frames of a sequence.

There are two different minimization algorithms available. The first assumes all boundary point to be fixed on the boundary and uses a LU decomposition of the discrete stiffness matrix to compute the minimum energy. The second one uses a conjugate gradient method and allows points to vary on the boundary curves. Points on straight lines may vary on the line, points on planar lines of symmetry may vary inside the plane. The type of boundary may be specified by the user when using a surface build with project *Build* inside the `*.bd` file otherwise the classification is done automatically: straight lines are straight, planar lines are assumed to be planar lines of symmetry and all other curves are assumed to be fixed. The CG algorithm must be used if a subsequent conjugation operation will be applied.

Conjuagtion can only be applied after a CG minimization step. After having applied a conjuagtion step by pressing `dual ...` the conjuagte harmonic surface may be displayed changing the task to `TASK: CONJUGATE`. For further information regarding the work with a surface and its conjugate see 7.2.3.

A good way to compute discrete minimal surfaces is to start with a rough discretization, minimize a certain number of steps and refine the triangulation from time to time depending on the triangulation quality. For this the project *Refine* may be used which provided several refine methods including curvature depending refinement.

`LOOPS-Ruler:`	Specify the number of loops for the minimization algorithms. Be careful: after having specified e.g. 50 the ruler does not reset later so you might be surprised when minimizating later again and again 50 loops.
`LU-step`, `LU-video`	Invoke the dirichlet minimizer using the LU decomposition for computing the minimum.
`CG-step`, `CG-video`	Invoke the CG minimizer, necessary for subsequent conjugation.
`dual step`, `dual video`	Apply the conjugation algorithm. The resulting surface is stored as the conjugate surface.
`regenerate`	Some surfaces make difficulties for the minimization algorithms especially when they have degenerate triangles. With this button such degenerated triangles can be removed, but be careful, the resulting triangulation may be singular and you will not be able to continue minimizing it.

7.2.8 Project Diri

Project *Diri* is composed of the projects *Build*, *Dual* and *Refine*. This enables you to create in an easy way initial surfaces, minimize them and apply conjugation. Also interactive refinement is available.

7.3 Misc Projects

7.3.1 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.

7.3.2 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.

7.3.2.1 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

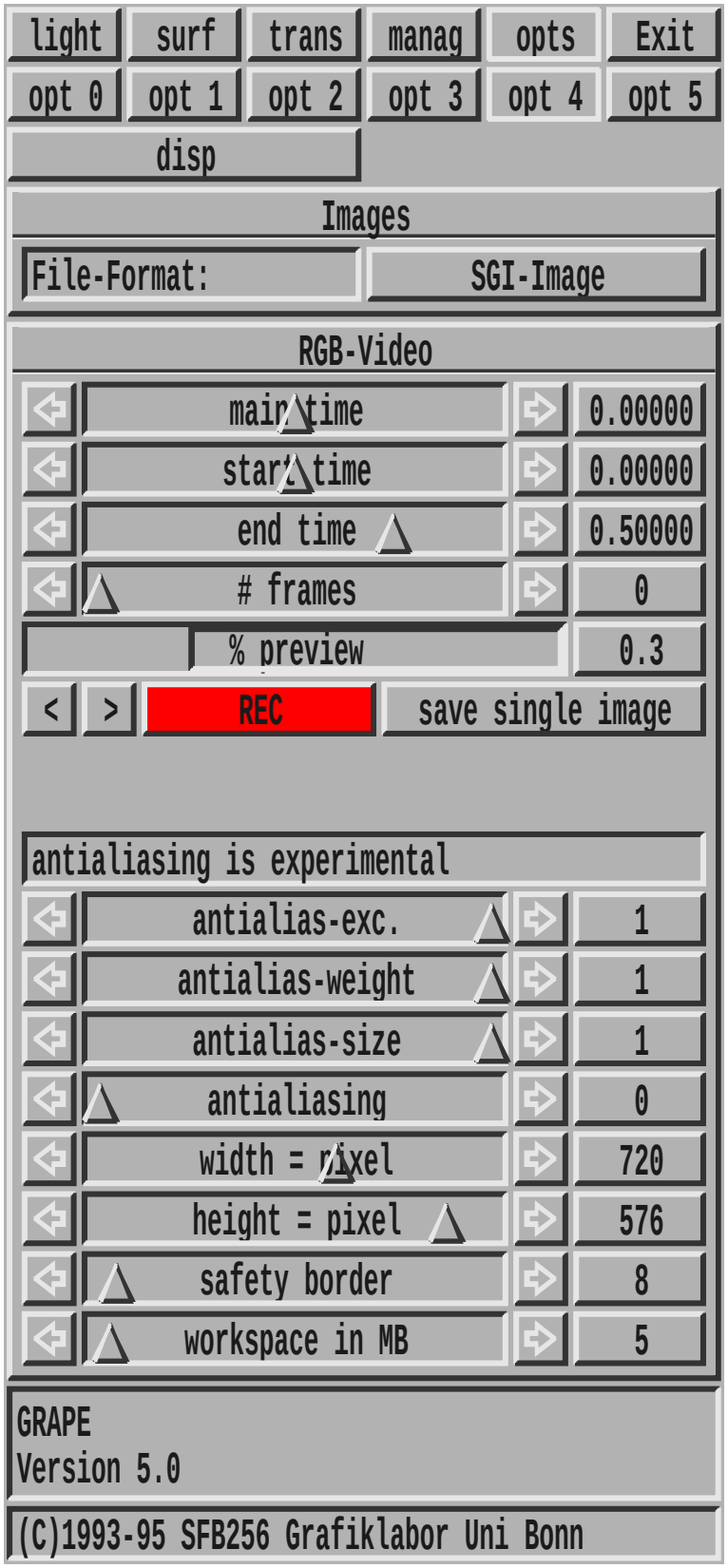


Figure 7.7: User Interface of Project RGBVideo, Containing the Images Part

`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.

7.3.2.2 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.

`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.

7.3.3 Video production and GRAPE

7.3.3.1 Overview

Currently we operate a SONY LVR-4000P video disc recorder that is being remote controlled from one of our SGI workstations doing visualization.

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.

For production of video tapes there is a JVC BR-S605E Professional S recorder also being controlled via its RS-232 interface.

7.3.3.2 A short guide for making your own movie

In this section we will assume that you already know about time scenes and function rulers. You should prepare a sequence that is played back when you press the PLAY button located below the Maintime ruler. For all operations described below we further assume that the video disc recorder is powered on, a **writable disc** is inserted and the remote/local switch on the left side of the front panel is in the **remote** position, i.e. the recorder is connected to an controlled by the workstation.

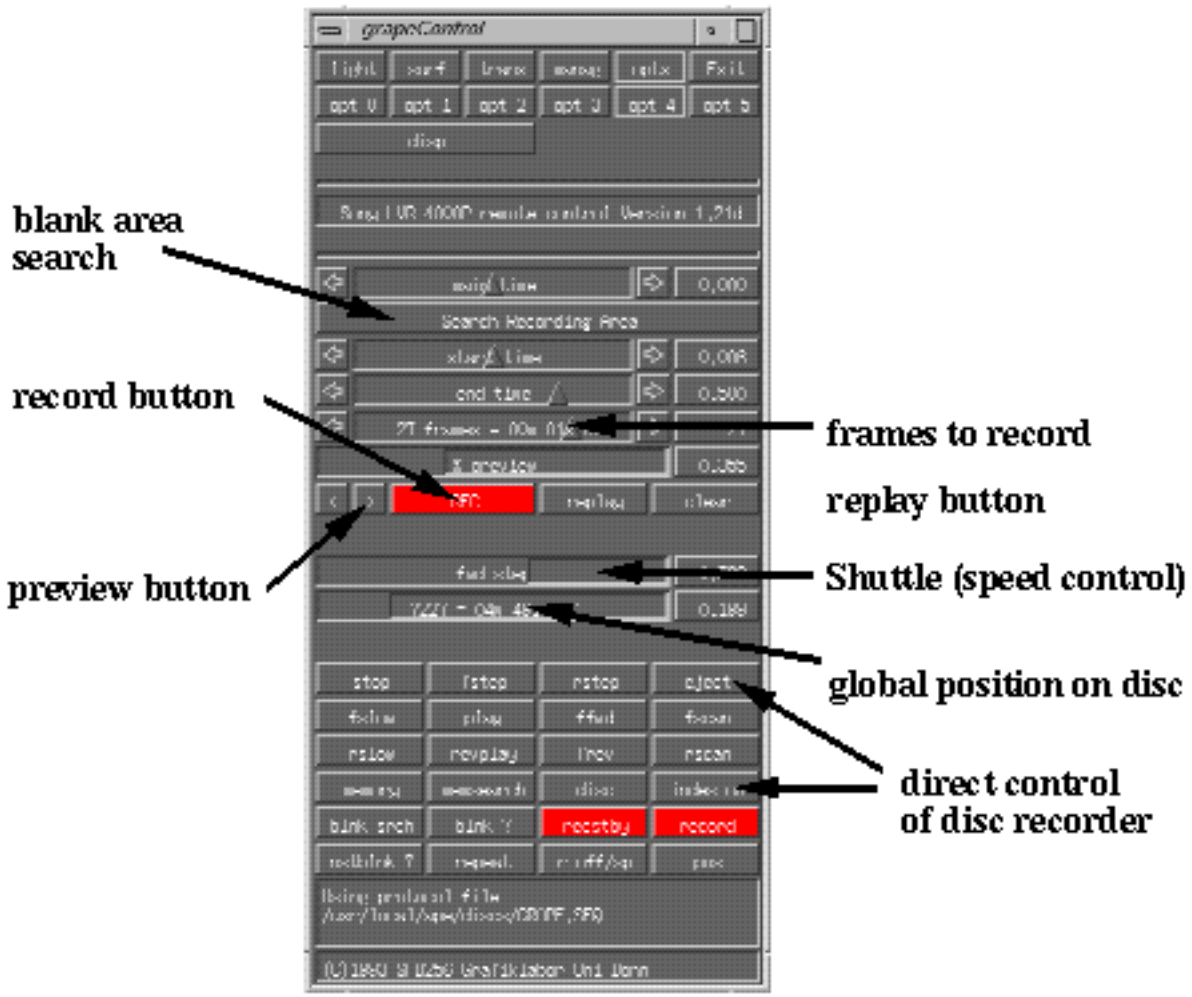
After having done this, you can start recording the movie. To do so, you should first set an environment variable before entering GRAPE:

```
setenv LVR_DEVICE /dev/ttyd1
```

or whatever device file your recorder might be connected to. The second preparation consists of starting the application that sends your workstation's screen to the video disc recorder. For the SGI Indigo2 with the Galileo board, this is

```
videoout
```

You may position the rectangle indicating the screen area to be recorded with the mouse. This should of course be inside the GRAPE graphics window. Having entered GRAPE you may load the project that controls the video disc recorder: chose CRVdisc or CRVdisc-rec, the latter does not provide full control of the recorder but show only those functions needed here. The buttons to control the recorder appear in the Options menu number four.



The next step is to perform a **blank area search**. The recorder we use writes onto an optical disc that can be written to just once. There is the notion of a **blank area**, that means those parts of the disc still available for recording. After pressing the **Search recording area** button it may take as long as one minute for the recorder to find out about those blank areas. After that, the recorder will remember them up to the next powerdown or change of media. If there are multiple blank areas on the disc, pressing the button multiple times will cycle through the possible recording areas.

Next, you should decide how long the film should be. The ruler labeled **# frames** controls how many frames will be written to disc. One second of video film is made up of 25 pictures. Remember, however, that you may slow down the replay speed of the video disc recorder while copying to VHS tape. Especially title screens should be recorded only on one frame, so the actual

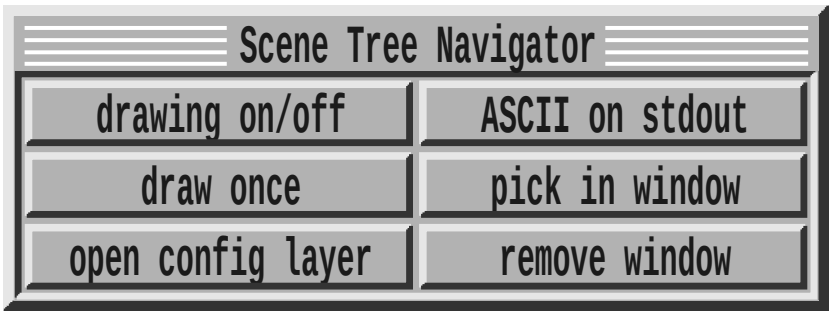


Figure 7.8: Main User Interface of Project Scene Tree

length once the film is ready will be set by the replay speed for that specific picture.

For convenience, the maintime ruler is also present in Options menu number four. After setting the start and end times, you may preview the sequence, the slider labeled **% preview** defines how many frames will be shown in order to save time. If all looks good, just press the big red **REC** button twice and recording will start.

If you have to stop the recording prematurely because something goes wrong, just press the middle mouse button until recording stops. You can then use the **clear** button to cancel standby mode of the recorder. Alternatively, put the remote/local switch already mentioned to the **local** position and then press the yellow **CLEAR** button on the recorder's front panel. Recording will stop immediately.

You then may review what has been recorded by pressing the **replay** button. For the next sequence to be recorded, just repeat the process.

The **Shuttle** slider simulates the recorder's speed control wheel: setting it to some position in the left half will reverse play the contents of the disc, the faster the more you set it to the left. To learn what speeds the recorder can realize with this knob, turn on **index mode.**

With the global position slider you can easily find positions on the disc: picture no. 1 is at the left, picture no. 36251 (the last one) at the right side.

7.3.4 Scene Tree

This project tries to display a global view of the data hierachy tree.

`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 the tree; the user interface operates with send–methods found at class `Root`.

Chapter 8

Interactive Classes and Devices

8.1 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 4.5.3) for some classes, but if you want to change an existing `Item` you have to use the methods.

8.1.1 General Structure

8.1.1.1 Menu concepts

The manager has got a list of layers (see 8.1.3.15), where each layer is a independent part of the manager. Each layer, which is a subclass of group (see 8.1.3.11), 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" 8.1.3.16 and "remove-layer" 8.1.3.16. 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 (8.1.3.11), 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 3.4.4), to the layer. If a different standard-menu is selected, for example the trans-menu (see 3.4.7), 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.

8.1.1.2 Adding- and Removing of Items

Each item must be added to a group or layer with the method "add-inter" (see 8.1.3.11) of the group or layer. The item can also be a whole group as it was described in the last subsubsection. The group will try to add the item with the size, which can be set using the method "set-pref-size" (see 8.1.3.14), and the position, which can be set using the method "set-pref-pos" (see 8.1.3.14). (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 8.1.3.11) and "remove-replacement" (see 8.1.3.11). You must remove the item always from the origin group, also if it was replaced, using

the method "remove-inter" (see 8.1.3.11). 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.

8.1.1.3 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 8.1.3.14) and "set-pref-size" (see 8.1.3.14), 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 8.1.3.14)). 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 8.1.3.11), 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 last subsection.) Pay attention, that this mechanism is different to the one used if you add item per item.

8.1.1.4 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.

8.1.1.5 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. 8.1.3.16) 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 8.1.3.14), 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 themself (the border, title and the place which isn't occupied by any element).

8.1.1.6 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 8.1.3.13). Normally this method is the "default-action". If you change this action method, you can change the behavior of the elements.

8.1.2 Class Hierarchy

Manager: connection between devices and GRAPE, receives event and initiates the associated actions, contains layer management and display method ruler management. (8.1.3.16 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. (8.1.3.14 Item)

- **Rectangle:** colored rectangle. (8.1.3.19 Rectangle)
- **Textmessage, StaticText:** static text and messages. (8.1.3.27 Textmessage, 8.1.3.25 Statictext)
- **Interactive:** abstract superclass for all interactive objects in the handle, e. g. clickable objects. Implements event handling. (8.1.3.13 Interactive)
  - **Input:** input field for numbers or single lines of text. (8.1.3.12 Input)
  - **Button:** starts action if clicked. (8.1.3.2 Button)
    - **XList_Button:** opens a layer containing an XList, FileList or Methlist.(8.1.3.29 XList_Button)
    - **Checkbox:** toggle a flag. (8.1.3.4 Checkbox)
    - **Cycle_Button:** button with switching label. (8.1.3.6 Cycle_Button)
    - **Step_Button:** autorepeating button. (8.1.3.26 Step_Button)
  - **Group:** rectangular container for Items, which forwards some methods to them. Since Group is a subclass of Item, Groups can be nested. (8.1.3.11 Group)
    - **Layer:** a movable overlay Group. (8.1.3.15 Layer)
    - **Bar1d:** abstract superclass for Ruler, Slider, Scrollbar. Edit scalar values with the mouse or by typed input. (8.1.3.1 Bar1d)
      - **Ruler, Slider:** ruler and slider. (8.1.3.20 Ruler, 8.1.3.22 Slider)
        - **Function_Bar1d:** abstract superclass for a Bar1d controlled by a spline. (8.1.3.8 Function_Bar1d)
        - **Function_Ruler, Function_Slider:** different looking versions of Function_Bar1d. (8.1.3.9 Function_Ruler, 8.1.3.10 Function_Slider)
      - **Scrollbar:** a scrollbar with bar size management. (8.1.3.21 Scrollbar)
    - **Spline_Editor:** interactive editor for splines. (8.1.3.24 Spline_Editor)
    - **XList:** Table of choosable strings and an input for arbitrary strings. (8.1.3.28 XList)
      - **FileList:** file selector. (8.1.3.7 FileList)
      - **MethList:** method selector for the class tree. (8.1.3.18 MethList)
    - **CheckField:** easy to use grouping class for Checkboxes, you can define complex connection rules. (8.1.3.3 CheckField)
    - **Color_Sel:** Color selector for RGB and HSV color model. (8.1.3.5 ColorSel)
    - **Matrix_Edit:** Interactive editor for some parts of transformation matrices. Used in trans-menu, light-menu, ... (8.1.3.17 Matrix_Edit)

Spline1d: Splines. (8.1.3.23 Spline1d)

8.1.3 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.

8.1.3.1 Bar1d

(subclass of 8.1.3.11 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,
  dfREGULAR_EXPRESSION
} 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 8.1.3.12 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 8.1.3.13 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.

8.1.3.2 Button

(subclass of 8.1.3.13 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 8.1.3.13). Normally this will be the "default-action" (8.1.3.13), which will send the method you can choose with "set-method" (8.1.3.13) to the instance you can choose using "set-instance" (8.1.3.13). Also the "default-action" will switch the button again. If you only set the method a button should send and set no instance, the button will send the method to the current object of the manager (see section 3.4.2). 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 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-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`.

8.1.3.3 CheckField

(subclass of 8.1.3.11 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 which 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, "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 bit-masks 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-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.

8.1.3.4 Checkbox

(subclass of 8.1.3.2 Button)

A Checkbox toggles a flag. Checkboxes can be combined to a CheckField which allows defining connection rules like radio-button-mode.

GRAPE(Checkbox, "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, "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.

8.1.3.5 Color_Sel

(subclass of 8.1.3.11 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`.

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, "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.

8.1.3.6 Cycle_Button

(subclass of 8.1.3.2 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, "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, "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 and show the correct label.

8.1.3.7 FileList

(subclass of 8.1.3.28 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`. Using the helper function `g_open_file_stddev (FILELIST *)` a Textdevice can be opened automatically and can be set as the standard Textdevice. The Textdevice must be closed at the end of the method using its `"delete"` method. If the FileList is implemented as an overlay menu in its own layer, the helper function `g_close_and_remove (FILELIST *)` will automatically close the Textdevice and 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 (g_open_file_stddev(filelist)) {
    ... /* Read data from the standard TextDevice */
  }
  else {
    /* User pressed "Cancel" or file doesn't exist  */
  }
  g_close_and_remove (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
sternFileList
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"`.

8.1.3.8 Function_Bar1d

(subclass of 8.1.3.22 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"`.

8.1.3.9 Function_Ruler

(subclass of 8.1.3.8 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.

8.1.3.10 Function_Slider

(subclass of 8.1.3.9 Function_Ruler)

GRAPE(Function_Slider, "new-instance")((see superclass method))

CLASS *Function_Slider
(see superclass method)

Overloading inherited method of class 8.1.3.8 Function_Bar1d.

8.1.3.11 Group

(subclass of 8.1.3.13 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-replacement")(new_replace)

GROUP *group
GROUP *new_replace

Add the group `"new_replace"` to the list of replaced groups. See 8.1.1.2 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, "delete")()

GROUP *group

First of all this group will send a delete method 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-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, "update")()

GROUP *group

Overloading inherited method of class 8.1.3.14 Item. Will call the `"update"` method of each interactive, which is contained in the group.

8.1.3.12 Input

(subclass of 8.1.3.13 Interactive)

An input allows you to view and edit `int`-, `float`-, `double`- and `char *`-variables.

```
typedef enum declaration_format{
  dfINT,
  dfFLOAT,
  dfDOUBLE,
  dfSTRING,
  dfREGULAR_EXPRESSION
} DECLARATION_FORMAT;
```

GRAPE(Input, "new-instance")(var, name, type, max_size)

CLASS *Input
void *var
char *name
DECLARATION_FORMAT type
int max_size

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 the SGI compiler 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.

8.1.3.13 Interactive

(subclass of 8.1.3.14 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 8.1.3.13.

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 8.1.3.14. 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-side-effect")(effect)

INTERACTIVE *interactive
INTERACTIVE *effect

Remove a side effect notifier.

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-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.

GRAPE(interactive, "switch")()

INTERACTIVE *interactive

Toggle between active and inactive state. Inactive Interactives are grayed out to prevent usage.

8.1.3.14 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 8.1.3.13 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`.

(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.

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 8.1.3.13

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-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 8.1.1.2.

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 8.1.1.2.

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.

8.1.3.15 Layer

(subclass of 8.1.3.11 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, "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 8.1.3.14 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 8.1.3.14 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`.

8.1.3.16 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 8.1.1.1 and in section 3.1.

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-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 8.1.1.2). 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 `Items` 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"`.

GRAPE(manager, "get-current-object")()

MANAGER *manager

Returns the current object.

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.

GRAPE(manager, "handle")(inst)

MANAGER *manager
INSTANCE *inst

Set `inst` as the current object, enter the interactive mode.

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 depended interactives, which are removed automatically when the display method isn't used any more.

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 4.4.6.

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-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, "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.

8.1.3.17 Matrix_Edit

(subclass of 8.1.3.11 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 are not documented up to now.

8.1.3.18 MethList

(subclass of 8.1.3.28 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.

8.1.3.19 Rectangle

(subclass of 8.1.3.14 Item)

A monochrome rectangle with 3D-looking border. There are no additional methods, see Item.

8.1.3.20 Ruler

(subclass of 8.1.3.1 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`.

8.1.3.21 Scrollbar

(subclass of 8.1.3.1 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`.

8.1.3.22 Slider

(subclass of 8.1.3.20 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.

8.1.3.23 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"`.

(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.

(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, "write")(file_name)

SPLINE1D *spline1d
char *file_name

Write itself as a `.sp` file.

8.1.3.24 Spline_Editor

(subclass of 8.1.3.11 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.

8.1.3.25 StaticText

(subclass of 8.1.3.27 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.

8.1.3.26 Step_Button

(subclass of 8.1.3.2 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");
```

8.1.3.27 Textmessage

(subclass of 8.1.3.14 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.

8.1.3.28 XList

(subclass of 8.1.3.11 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-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, "new-instance")(meth, inst, name)

CLASS *XList
char *meth
INSTANCE *inst
char *name

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.

8.1.3.29 XList_Button

(subclass of 8.1.3.2 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 deleted 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"`.

8.1.4 Utility Functions

```
int 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 `G_MSGBOX_BUTTON_OK`, `G_MSGBOX_BUTTON_ABORT`, `G_MSGBOX_BUTTON_CANCEL`, `G_MSGBOX_BUTTON_YES` and `G_MSGBOX_BUTTON_NO`. If buttons is 0 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 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 active as long as "dismiss" is not pressed.

8.2 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
CLASS *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")();
```

8.2.1 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:

```
typedef struct {
    ...
    /* superclass structure */
    unsigned RW_flags;          /* read/write possible? */
} DEVICE;
```

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
```

8.2.2 GraphicDevice

The class GraphicDevice is only a superclass for the different classes representing the different hardware platforms. Recently implemented subclasses are

- GraphicGt: GL
- GraphicPs: Postscript (to create a 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.
- GraphicX11: X-Windows
- GraphicVideo: Videorecorder, subclass of GraphicGt

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. The structure of the class GraphicDevice is defined as follows:

```
typedef struct graphicdevice{
 ...
 /* superclass structure */
 int *info;            /* pointer to device specific informations */
 void (*update)(void);
 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);
 void (*draw_direct)(VEC3);
 void (*begin_patch_direct)(void);
 void (*patch_vertex_direct)(VEC3);
 void (*patch_normal_direct)(VEC3);
 void (*patch_color_direct)(VEC3);
 void (*end_patch_direct)(void);
 int  (*generate_lightsource)(int);
 void (*lightsource)(int, int, LIGHTSOURCE_DEV *);
 void (*delete_lightsource)(int);
 int  (*generate_clipplane)(void);
 void (*clipplane)(int, int, CLIPPLANE_DEV *);
 void (*delete_clipplane)(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 */
} 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")();

This pointer has to be updated if the standard device might have changed. The variables must not be changed by hand an even the function pointers can change when switching methods are called.

Now a brief description for the primitive graphic routines:

8.2.2.1 Primitive Graphic Routines

void update();

All data will be put on the screen including buffered data. Therefore the screen will display the actual picture.

void clear();

The whole screen will display the actual background color (see `dev ->attribute()` below on 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 below.

```
/* modes */
G_MODE_GET
G_MODE_SET
G_MODE_PRE
G_MODE_POST

/* types */
G_MATRIX_NONE
G_MATRIX_PROJECT
G_MATRIX_VIEW
G_MATRIX_MODEL
G_MATRIX_PVM
G_MATRIX_IVM
```

void transform(int mode, int type, MATRIX44 matrix);

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. The transponent of

IVM transforms the normal vectors.

void move(VEC3 point);

The 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();

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.

void end_patch();

This closes the definition of the actual patch and the patch is displayed.

int generate_lightsource(type);
int type == G_LIGHT_AMBIENT, DIRECT, POINT, SPOT;

```
G_LIGHT_AMBIENT
G_LIGHT_DIRECT
G_LIGHT_POINT
G_LIGHT_SPOT
```

This defines a new light source. The type's number may be 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 (=1) or off (=0) ? */
    float color[3];     /* rgb light color (incl. intensity) */
    float position[3];  /* POINT | SPOT light source position */
    float direction[3]; /* DIRECT | SPOT light direction */
    float open_angle;   /* SPOT open angle */
} LIGHTSOURCE_DEV;
```

void lightsource(int mode, int light, LIGHTSOURCE_DEV *data);

Read or define any light source. The light's number light has to be created by a previous `dev ->generate-lightsource()`. A light source is defined its `data`, its type can not be changed by `G_MODE_SET` but it is defined by `dev ->generate-lightsource()`.

void delete_lightsource(int light);

This removes the given light source previously defined by dev ->generate-lightsource().

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.

8.2.2.2 GraphicDevice attribute() parameters: modes, attribute- and datatypes

Now the definition of the attribute()-function:

int attribute(mode,type,data);

int mode == G_MODE_QUERY, GET, SET, SWAP;
int type == G_RASTER_ABILITY, ... ;
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.

```
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 "doublebuffer" 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
```

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         ---            hardware 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          {float aspect}    win_size_x / win_size_y
G_BACKGROUND_COLOR  GET | SET    {float rgb[3]}
G_LINE_COLOR        GET | SET    {float rgb[3]}
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    {float rgb[3]}   (only X11)
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
```

The table above used the following typedefs:

```
int on_off, yes_no;          /*  no/off=0  yes/on=1 */

typedef struct suprop_dev {
 float emission[3];  /* rgb color emission (incl. intensity) */
 float ambient[3];   /* ambient rgb reflectance              */
 float diffuse[3];   /* diffuse rgb reflectance              */
 float specular[3];  /* specular rgb reflectance             */
 float specular_exp; /* specular reflectance exponent        */
 float transparency; /* transparency factor (0..1)           */
} SUPROP_DEV;

typedef struct depthcue_type {
 int on_off; /* off=0, on=1 */
 float zmin; /* dist. where depth cueing begins (full intensity) */
 float zmax; /* dist. where depth cueing ends (reduced intensity)*/
 float dark; /* max. reduction of intensity (0.0 <= dark <= 1.0) */
} DEPTHCUE_TYPE;

typedef struct text_dev {
 int font;   /* text font; default = 1   (device dependent) */
 float size; /* text size; default = 1.0 (device dependent) */
 float angle;/* text angle in degrees; 0.0 = horizontal direction */
} TEXT_DEV;
```

8.2.2.3 Methods on GraphicDevice

(GRAPHICDEVICE *)GRAPE(dev1, "copy")(dev2, name)

GRAPHICDEVICE *dev1
GRAPHICDEVICE *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`.

GRAPE(dev, "write")()

GRAPE(dev, "read")()

GRAPHICDEVICE *dev

The methods `"read"` and `"write"` get or put the important graphic data of GraphicDevice `dev` from or on the actual text device. These data are the transformation matrices, light, color and surface properties which can be affected by the interactive part of **GRAPE**. Therefore a selected status can be saved with `"write"` and restored with `"read"`.

(int)GRAPE(dev, "grid-patch")(int)

GRAPE(dev, "set-grid-patch")(mode)

GRAPHICDEVICE *dev
int mode

Any graphic device has two different modes. If patches are displayable they will be displayed shaded with normals at the vertices or displayed flat without normal information. Otherwise only wireframes are displayed.

The current mode is hold in the device instance variable `grid_patch`. The values are `G_GRID` for wireframes 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.

GRAPE(dev, "get-pick-pos")(pickx, picky, pickkey)

GRAPHICDEVICE *dev
float *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. 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.

8.2.3 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);
}
```

8.2.4 GraphicVideo

The class GraphicVideo is a subclass of GraphicGt. It allows to transfer a sequence of single pictures created on an Iris 4D/VGX to a video recorder. To accomplish this an instance of class GraphicVideo has to be created and set to the standard device. This assures the recording of the screen display after every doublebuffer swap (8.2.2.2). The type definition of class GraphicVideo is identical to GraphicGt.

(GRAPHICVIDEO *)GRAPE(GraphicVideo, "new-instance")(name)

CLASS *GraphicVideo
char *name

The method "new-instance" creates a new instance of class GraphicVideo.

GRAPE(dev, "delete")()

GRAPHICVIDEO *dev

The method "delete" removes the given instance of class GraphicVideo. If a recording session was in progress this method restores the screen with the command genlock(0).

GRAPE(GraphicVideo, "start-video")()

The method "start-video" defines the beginning of the recording session, i.e. every doublebuffer swap invokes the command sequence to record the current picture on the screen. One option allows to simulate the session to get an idea of the film.

GRAPE(GraphicVideo, "config-video")(nr-of-frames)

int nr-of-frames

The method "config-video" defines the number of frames recorded for of the same screen picture, e.g. if a scenery should be visible in the film for about ten seconds and there are 25 frames per second the variable nr-of-frames has to be set to 250 to accomplish this aim.

8.2.5 ControlDevice

The class ControlDevice is only a superclass for the different classes representing the different hardware platforms. Recently implemented subclasses are

- ControlX11: X Window System, Version 11, Release 4

Up to now only the X11 ControlDevice is implemented because all supported machines support X11. Though it is easy to port.

The class ControlDevice defines all pointers to the primitive functions containing the control output routines. The instance variable info points to a machine dependent structure containing the device's static variables. The structure of the class ControlDevice is defined as follows:

```
typedef struct controldevice{
 ...
 /* superclass structure */
 int   *info;          /* pointer to device specific information */
 void  (*update)(void);
 int   (*attribute)(int, int, int *);
 void  (*clear)(void);
 void  (*set_color)(int);
 void  (*set_rgb_color)(VEC3);
 void  (*set_text_color)(int);
 void  (*get_color)(int *);
 void  (*change_color)(int, VEC3);
 void  (*moveto)(double, double);
 void  (*drawto)(double, double);
 void  (*relative_draw)(double, double);
 void  (*draw_string)(double, double, char *);
 void  (*get_mouse)(double *, double *);
 void  (*fix_mouse)(double, double, double, double, double, double);
 void  (*free_mouse)(void);
 void  (*set_mouse)(double, double);
 void  (*rectangle)(double, double, double, double);
 void  (*circle)(double, double, double);
 void  (*reinit)(void);
 void  (*get_control_size)(double *, double *);
 void  (*wait)(double);
 void  (*clear_event_queue)(void);
```

```
  void  (*get_spaceball_data)(double *, double *, double *, double *,
                              double *, double *, double *, int *);
  int   (*handle_events)(EVENT *, int);
  int   (*menu_button)(void);
  int   (*read_string)(double, double, char *, double);
  int   (*middle_button)(void);
  int   (*get_free_color_index)(void);
  G_CLIP_DESCRIPTOR *
        (*kind_of_clipping)(void);
  void  (*set_clip_rects)(GRECT_LIST *);
  double (*get_unit_per_pixel)(void);
  double (*stringwidth)(char *);
  double (*stringheight)(char *);
 } CONTROLDEVICE;
```

A program has access to the primitive control routines for the actual device if it demands a pointer to the standard device by the command line

ctl = (CONTROLDEVICE *)GRAPE(ControlDevice, "get-stddev")();

This pointer has to be updated if the standard device might have changed. Now a brief description for the primitive control menu routines (especially the implementation for the X Window System):

8.2.5.1 Primitive Control Routines

void update();

All data will be put on the screen including buffered data. Therefore the screen will display the actual menu.

void clear();

The whole control window will display the actual background color (see ctl ->attribute() below on how to set the background color).

```
/* modes */
G_MODE_GET
G_MODE_SET
G_MODE_SWAP

/* types */
G_DOUBLEBUFFER
G_HAS_DOUBLEBUFFER
```

int attribute(int mode, int type, void *data);

Get or set an attribute. The data item has to be casted to the appropriate type. For the X11 control menu there are currently only two types supported. Mode `G_MODE_GET` on type `G_DOUBLEBUFFER` informs if doublebuffer mode is supported and mode `G_MODE_GET` on type `G_HAS_DOUBLEBUFFER` specifies if this control device has a doublebuffer. Other functions may be added, if necessary.

void moveto(double x, double y);

The pen position is set to the coordinate (x, y).

void drawto(double x, double y);

Draws a line from the current pen position to the position (x, y) and updates the pen position. The line regards color settings.

void relative_drawto(double dx, double dy);

Draws a line from the current pen position to the position (dx, dy) relative to the current position. If the current position is (x, y) the new position is (x + dx, y + dy). The pen position will be updated and the line regards color settings.

void draw_string(double x, double y, char *string);

Draws the text given in string at the position (x, y) on the control menu with the current text color and font. The text color can be set by ctl ->set-text-color().

void set_color(int color);

According to the given colormap index color the current color for drawing is set. The first sixteen colormap entries are hardcoded on color screens to black, red, green, yellow, blue, magenta, cyan, white, white (`FOREGROUND_COLOR`), gray40 (`BACKGROUND_COLOR`), gray70 (`PRESSED_LOWER_COLOR`), gray20 (`PRESSED_UPPER_COLOR`), gray90 (`TEXT_COLOR`), gray10 (`LINE_COLOR`), goldenrod (`LIGHT_ON_COLOR`). The last one is still unused. On black and white screens the colors are substituted appropriately. To use the remaining entries you have to ask for a free colormap index and change the color which is at first undefined. If you do not succeed to allocate a cell on small systems black is used.

void set_text_color(int color);

According to the given colormap index color the current color used for drawing text is set.

void get_color(int *color);

This function returns the index to the current colormap entry used for drawing.

void change_color(int color, VEC3 rgb);

The current color is set to `color` and the color map entry is set to `rgb`. On small systems the corresponding color cell is changed. On black and white terminals the `color` is set to white if the sum of the `rgb` values are greater than 1.5 otherwise black.

int get_free_color_index();

The index to the next free colormap entry is returned otherwise zero. On small color systems a read write color cell is allocated. If it fails this entry remains black.

There are some more primitive functions dealing with mouse control, keyboard input, draw circles or rectangles, resizing, reinitialization and handle events. If you are interested see source listings.

8.3 Text Devices

8.3.1 Text-Devices

To provide machine independent character output **GRAPE** includes the class TextDevice which is a subclass of Device. There are other instances of class Textdevice to simplify file input/output.

The written files are machine-dependent because of different binary encoding. An XDR-Device is under development which will solve this problem.

(TEXTDEVICE *)GRAPE(dev, "set-stddev")()

(TEXTDEVICE *)GRAPE(TextDevice, "get-stddev")()

(FILE *)GRAPE(dev, "get-file-pointer")()

TEXTDEVICE *dev

Every character input/output is always performed on the actual textdevice or standard textdevice (stddev). The method "get-stddev" on class Textdevice supplies a pointer to the instance of the standard device which is `/dev/tty` unless it is explicitly changed. Therefore every character output is send to the screen and input is read from the keyboard. The standard device can be changed with "set-stddev", this method's return code is the pointer to the old standard device.

To transfer data to or from a file there exist the following classes:

- InputFile: file only for reading
- OutputFile: file only for writing
- InOutFile: if file exists opened according to filemode else created
- Terminal: Terminal (with cursor etc.) (not yet implemented)
- TextWindow: Textwindow on graphicterminals (not yet implemented)

Their structure is identical to Textdevice, but only the corresponding pointers to the primitive text functions are provided (see below). As an example for how to handle a file the following commands are implemented for an input file:

(INPUTFILE *)GRAPE(InputFile, "new-instance")(name)

GRAPE(dev, "delete")()

CLASS *InputFile
INPUTFILE *dev
char *name

The first command opens an existing file name. A new instance of type InputFile is created. The second command closes the appropriate file and removes the instance. To make a file accessible to the read and write methods on the standard textdevice send the method "set-stddev" to this instance (GRAPE(dev, "set-stddev")();). On successful execution all functions return a value of type (TEXTDEVICE *) unequal to NULL.

Moreover an actual error device exists which is used by the error handling. The appropriate methods provide the same features as the standard textdevice methods. The special filepointer NULL is handled as well:

(TEXTDEVICE *)GRAPE(dev, "set-error-device")()

(TEXTDEVICE *)GRAPE(TextDevice, "get-error-device")()

TEXTDEVICE *dev

8.3.2 Primitive Text Functions

For efficiency the class TextDevice contains some primitive functions for text input/output to files. An instance of TextDevice contains pointers to the C functions which implements these primitive functions. But for an InputFile only the pointers to seek and read functions are available. Make sure that the pointer to the standard device is up to date so that the primitive functions are accessible.

Note that because of the implementation via C functions the created files are machine dependent. Whenever possible the **GRAPE** libraries are compiled using **ieee** floating point format and big endian encoding for integers to achieve maximum compatibility.

An XDR-Device is under development which will solve this problem.

The structure of class TextDevice is defined as follows:

```
typedef struct {
  DEVICE device;
  FILE *fp;             /* FILE pointer */
  int term_flags;       /* internal use only; do not modify these. */
  void (*query)();      /* should do something in future */

/* seek functions to be called from C (quick) */
  int (*seek ) ( long offset, int from_where_to_start );
  int (*tell ) ( long *offset );

/* write functions to be called from C (quick) */
  int (*write_char   ) ( char c);
  int (*write_bytes  ) ( int n_bytes, char *ptr );
  int (*write_string ) ( char *s );       /* incl '\\0' !! */
  int (*write_short  ) ( short x);
  int (*write_int    ) ( int i);
  int (*write_long   ) ( long l);
  int (*write_float  ) ( float f);
  int (*write_double ) ( double d);

/* read functions to be called from C (quick) */
/*  WARNING: The value is read into the pointer variable supplied !!
            rc == 0 ==> error;     rc != 0 ==> ok   */
  int (*read_char   ) ( char *c_ptr );
  int (*read_bytes  ) ( int n_bytes, char *ptr );
  int (*read_string ) ( char *s );       /* incl. '\\0' !! */
  int (*read_short  ) ( short *x_ptr );
  int (*read_int    ) ( int *i_ptr );
  int (*read_long   ) ( long *l_ptr );
  int (*read_float  ) ( float *f_ptr );
  int (*read_double ) ( double *d_ptr );
} TEXT DEVICE;
```

Now a brief description of the primitive functions.

```
int (*seek)(long offset, int from\_where\_to\_start);
```

seek looks for a certain position in a file (as fseek in C). Legal values for `from_where_to_start` are zero, one and two to start searching at the beginning, the current position or at the end of the file. The value `offset` supplies the relative position to be seeked.

```
int (*tell ) ( long *offset );
```

The variable *offset* will contain the actual position relative to the file's beginning. The return code is zero, if an error occurs.

The primitive function's return code is always zero, if an error occurs.

```
int (*write_char   ) ( char c);
```

The character *c* will be written into the file.

```
int (*read_char   ) ( char *c_ptr );
```

One character is read into the variable, which is addressed by the pointer. At end of file or any other error, the return code is zero.

```
int (*write_bytes  ) ( int n_bytes, char *ptr );
```

The amount of *n_bytes* beginning at position *ptr* is written into the file.

```
int (*read_bytes  ) ( int n_bytes, char *ptr );
```

The amount of *n_bytes* is read into memory beginning at position *ptr*. Assure to allocate enough memory.

```
int (*write_string ) ( char *s );       /* incl '\0' !! */
```

The string s is written into the file.

```
int (*read-string)(char *s);       /* incl. '\0' !! */
```

The string *s* is written into the file.

```
int (*read_string ) ( char *s );       /* incl. '\0' !! */
```

The string *s* is read from the file until the end character is encountered. Assure to allocate enough memory.

```
int (*write_short  ) ( short x);
int (*write_int    ) ( int i);
int (*write_long   ) ( long l);
int (*write_float  ) ( float f);
int (*write_double ) ( double d);
```

These functions write one variable into the file.

```
int (*read_short  ) ( short *x_ptr );
int (*read_int    ) ( int *i_ptr );
int (*read_long   ) ( long *l_ptr );
int (*read_float  ) ( float *f_ptr );
int (*read_double ) ( double *d_ptr );
```

These functions read one variable from the file into memory addressed by the appropriate pointer.

Advice: these files are in general not transferable between different computers, because of different binary coding.

An XDR-Device is under development which will solve this problem.

Chapter 9

Utility Classes and Functions

9.1 Utility Classes

9.1.1 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;

typedef struct g_list {
  /* INSTANCE */
  CLASS *class;
  char *name;
  INSTANCE *next;
  unsigned int refcount;
  /* G_LIST */
  G_NODE *head;   /* first element */
  G_NODE *tail;   /* last element */
  G_NODE *iter;   /* current element for iteration functions */
} G_LIST;
```

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_Lists are used for several internal GRAPE lists, for example all information about classes, methods and items in a project is stored in this way.

9.1.1.1 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, "delete")()

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, "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_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. 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_remove(glist, obj)

G_LIST *glist
void *obj

Remove an object from the list. `g_list_rem_head` and `g_list_rem_tail` remove the first resp. last 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);
  }
```

9.1.1.2 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.

9.1.2 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 (5.3.4), and for storing information about the connectivity of triangulations (see `->`connectivity methods on Triang2d (5.3.1.4)).

```
typedef struct list_of_inst {
    /* SYSTEM */
    CLASS *class;
    char *name;
    INSTANCE *next;
    unsigned int refcount;
    /* USER */
    int num_of_objects;
    int max_num_of_objects;
    int *flags;
    INSTANCE **objects;
} LIST_OF_INST;
```

9.1.2.1 Memory Methods on List-Of-Inst

GRAPE(list, "delete")()

`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.

9.1.2.2 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")()

```
LIST_OF_INST *list
```

Create info `list` entries for a List_Of_Inst instance. Entries are class, name, refcount, number of objects and class and flag for each object.

9.2 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 (`VEC3`, `DVEC4`, and n_dim vectors),
- matrix operations and
- methods for List_Of_.... handling.

Furthermore a number of complex functions that are available in **GRAPE** simplify the handling of complex variables.

9.2.1 Memory Functions

All memory functions are based on the functions `mem_alloc` and `mem_free` which are described in section 4.2.2. 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(sizet 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.

9.2.1.1 Functions for allocating/freeing memory of data-arrays

```
char *char_alloc(int nr_of_char)
char *char_free(char *ptr, int nr_of_char)
int *int_alloc(int nr_of_int)
int *int_free(int *ptr, int nr_of_int)
float *float_alloc(int nr_of_float)
float *float_free(float *ptr, int nr_of_float)
double *double_alloc(int nr_of_double)
double *double_free(double *ptr, int 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 retun a NULL-pointer. The free-functions will free a memory bloxk which is big enough for nr_of_... elemnts at the adress ptr points to. The free functions will return a NULL-pointer.

9.2.1.2 Functions for allocating/freeing memory of pointer-arrays

```
float **float_p_alloc(int nr_of_float_p)
float **float_p_free(float **ptr, int nr_of_float_p)
double **double_p_alloc(int nr_of_double_p)
double **double_p_free(double **ptr, int nr_of_double_p)
char **char_p_alloc(int nr_of_char_p)
char **char_p_free(char **ptr, int 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.

9.2.1.3 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(int nr_of_int2)
INT2 *int2_free(INT2 *ptr, int nr_of_int2)
INT2 **int2_p_alloc(int nr_of_int2_p
INT2 **int2_p_free(INT2 **ptr, int nr_of_int2_p)
VEC3 *vec3_alloc(int nr_of_vec3)
VEC3 *vec3_free(VEC3 *ptr, int nr_of_vec3)
VEC3 **vec3_p_alloc(int nr_of_vec3_p)
VEC3 **vec3_p_free(VEC3 **ptr, int nr_of_vec3_p)
DVEC3 *dvec3_alloc(int nr_of_dvec3)
DVEC3 *dvec3_free(DVEC3 *ptr, int nr_of_dvec3)
DVEC3 **dvec3_p_alloc(int nr_of_dvec3_p)
DVEC3 **dvec3_p_free(DVEC3 **ptr, int nr_of_dvec3_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(int nr_of_datatype)
datetype_free(datatype *ptr, int nr_of_datatype)
datatype_p_alloc(int nr_of_datatype_p)
datatype_p_free(datatype **ptr, int nr_of_datatype_p)
```

9.2.2 Functions on VEC3

VEC3 is declared as **typedef float VEC3[3];** and the following functions are available:

`void` **g_matrix44_mult_vec3** `(MATRIX44 m, VEC3 v)`
multiply `v` with the matrix `m`

`double` **g_vec3_abs** `(VEC3 v)`
return 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)**
return 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`

`double` **g_vec3_dist** `(VEC3 p, VEC3 q)`
returns the distance of the two points in R3

`int` **g_vec3_get_normal** `(VEC3 u, VEC3 v, VEC3 w)`
`u` = a unit vector orthogonal on `v` and `w`

`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_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`

`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_length** `(VEC3 v, double r)`
calculates a vector of length `r` from `v`, replacing `v`

`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`

9.2.3 Functions on DVEC3

DVEC3 is declared as **typedef double DVEC3[3];** and the following functions are available:

`void` **g_matrix44_mult_dvec3** `(MATRIX44 m, DVEC3 v)`
multiply `v` with the matrix `m`

`double` **g_dvec3_abs** `(DVEC3 v)`
return the length of `v`

`void` **g_dvec3_add** `(DVEC3 vadd, DVEC3 v, DVEC3 w)`
`vadd` = `v` + `w`

`void` **g_dvec3_assign** `(DVEC3 v, DVEC3 w)`
`v` = `w`

`double` **g_dvec3_circle** `(DVEC3 center, DVEC3 p[3])`
calculates the center of the circle through `p[0]`, `p[1]` and `p[2]` and returns the radius

`void` **g_dvec3_crossprod** `(DVEC3 u, DVEC3 v, DVEC3 w)`
`u` = cross product of `v` and `w`

`float` ***g_dvec3_cv_vec3** `(VEC3 v, DVEC3 w)`
`v` gets the values of `w` and `v` is returned

`double` ***g_vec3_cv_dvec3** `(DVEC3 v, VEC3 w)`
`v` gets the values of `w` and `v` is returned

`double` **g_dvec3_dist** `(DVEC3 p, DVEC3 q)`
returns the distance of the two points in R3

`int` **g_dvec3_get_normal** `(DVEC3 u, DVEC3 v, DVEC3 w)`
`u` = a unit vector orthogonal on `v` and `w`

`int` **g_dvec3_get_normal_to_plane** `(DVEC3 u, DVEC3 p, DVEC3 q, DVEC3 r)`
`u` = the normal at `p` of the triangle `p_q_r`

`double` **g_dvec3_mult** `(DVEC3 v, double r)`
multiplies `v` by the number `r`

`int` **g_dvec3_normalize** `(DVEC3 v)`
calculates a unit vector from `v`, replacing `v`

`int` **g_dvec3_proj_onto_line** `(DVEC3 v, DVEC3 u)`
replaces `v` by its projection on span (`u`) (if `u` has unit length)

`int` **g_dvec3_proj_onto_plane** `(DVEC3 v, DVEC3 u)`
replaces `v` by its projection on the plane with normal `u`

`int` **g_dvec3_set_length** `(DVEC3 v, double r)`
calculates a vector of length `r` from `v`, replacing `v`

`double` **g_dvec3_skp** `(DVEC3 v, DVEC3 w)`
returns the scalar product of `v` and `w`

`double` **g_dvec3_sqrabs** `(DVEC3 v)`
returns the square length of `v`

`double` **g_dvec3_sqrdist_sign** `(DVEC3 p, DVEC3 q)`
returns the square distance of two points

`void` **g_dvec3_sub** `(DVEC3 vsub, DVEC3 v, DVEC3 w)`
`vsub` = `v - w`

9.2.4 Functions on DVEC4

DVEC4 is declared as **typedef double DVEC4[4];**

Some of the functions for 4-dimensional vectors depend on signature. Signature is a DVEC4 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_dvec4** `(MATRIX44 m, DVEC4 v)`
multiplies `v` by the matrix `m`, replacing `v`

`void` **g_matrix44_mult_dvec4_dvec4** `(MATRIX44 m, DVEC4 v, DVEC4 w)`
multiplies `v` by `m` and stores the result in `w`

`double` **g_dvec4_abs** `(DVEC4 v)`
returns the length of `v`

`double` **g_dvec4_abs_sign** `(DVEC4 v, DVEC4 sign)`
returns the length of `v` (depending on signature)

`void` **g_dvec4_add** `(DVEC4 vadd, DVEC4 v, DVEC4 w)`
`vadd` = `v` + `w`

`double` **g_dvec4_angle** `(DVEC4 p, DVEC4 q1, DVEC4 q2, DVEC4 sign)`
returns the angle at `p` in the triangle `q1-p-q2` (depending on signature)

`void` **g_dvec4_assign** `(DVEC4 v, DVEC4 w)`
`v` = `w`

`double` **g_dvec4_ctg** `(DVEC4 p, DVEC4 q1, DVEC4 q2, DVEC4 sign)`
returns the cotangens of the angle at p in the triangle `q1-p-q2` (depending on signature)

`double` **g_dvec4_dist** `(DVEC4 p, DVEC4 q)`
returns the distance of the two points

`double` **g_dvec4_dist_sign** `(DVEC4 p, DVEC4 q, DVEC4 sign)`
returns the distance of two points (depending on signature)

`void` **g_dvec4_list_assign** `(DVEC4 *v, DVEC4 *w, int nelem)`
copies the list of vectors `w` to the list `v`

`void` **g_dvec4_mult** `(DVEC4 v, double r)`
multiplies each coordinate of `v` by `r`, replacing `v`

`int` **g_dvec4_normalize** `(DVEC4 v)`
creates a unit vector from `v`, replacing `v`

`int` **g_dvec4_normalize_sign** `(DVEC4 v, DVEC4 sign)`
creates a unit vector from `v` (depending on signature)

`int` **g_dvec4_set_length** `(DVEC4 v, double r)`
creates a vector of length `r` from `v`, replacing `v`

`int` **g_dvec4_set_length_sign** `(DVEC4 v, double r, DVEC4 sign)`
creates a vector of length `r` from `v` (depending on signature)

`double` **g_dvec4_skp** `(DVEC4 v, DVEC4 w)`
returns the scalar product of `v` with `w`

`double` **g_dvec4_skp_sign** `(DVEC4 v, DVEC4 w, DVEC4 sign)`
returns the scalar product of `v` and `w` (depending on signature)

`double` **g_dvec4_sqrabs** `(DVEC4 v)`
returns the square of the length of `v`

`double` **g_dvec4_sqrabs_sign** `(DVEC4 v, DVEC4 sign)`
returns the square length of `v` (depending on signature)

`double` **g_dvec4_sqrdist** `(DVEC4 p, DVEC4 q)`
returns the square of the distance of the two points

`double` **g_dvec4_sqrdist_sign** `(DVEC4 p, DVEC4 q, DVEC4 sign)`
returns the square distance of two points (depending on signature)

`void` **g_dvec4_sub** `(DVEC4 vsub, DVEC4 v, DVEC4 w)`
`vsub = v - w`

9.2.5 Matrix Functions

Matrices are declared as float or double arrays:
typedef float MATRIX44[4][4];
or
typedef double DMATRIX44[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`

`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_dmatrix44_mult_dvec4** `(DMATRIX44 m, DVEC4 v)`
`v = m * v`, where `v` is of the type `VEC3` or `DVEC4`

`void` **g_dmatrix44_mult_dvec4_dvec4** `(DMATRIX44 m, DVEC4 v, DVEC4 w)`
`w = m * v`

`void` **g_matrix44_set_identity** `(MATRIX44 m)`
`m` = identity matrix (also available on DMATRIX44)

`void` **g_matrix44_transp** `(MATRIX44 m, MATRIX44 t)`
transposes `m` and stores result in `t` (also available on DMATRIX44)

9.2.6 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`

9.2.7 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)`

9.2.8 String Functions

The GRAPE memory manager prohibits to use standard string functions which allocate memory using `malloc`. So we designed some utility functions:

`char *` **g_strdup** `(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, 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`.

9.2.9 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)`

9.2.10 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

9.2.11 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.

`float` **g_sqrtf** `(float x)`

`float` **g_fabsf** `(float x)`

`float` **g_powf** `(float x, float y)`

`float` **g_cosf** `(float x)`

`float` **g_sinf** `(float x)`

`float` **g_tanf** `(float x)`

`float` **g_acosf** `(float x)`

`float` **g_asinf** `(float x)`

`float` **g_atanf** `(float x)`

`float` **g_atan2f** `(float x)`

`float` **g_floorf** `(float x)`

`float` **g_fmodf** `(float x, float y)`

`float` **g_ceilf** `(float x)`

`float` **g_expf** `(float x)`

`float` **g_logf** `(float x)`

`float` **g_log10f** `(float x)`

Without the appendix `f` all these functions are available on `double` as well. For example:

`double` **g_sqrt** `(double 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`

9.2.12 Functions for Triang2d

For the `TRIANG2D` structure see 5.3.1.

`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`.

9.2.13 Complex Functions

In the following we list a number of complex functions available in **GRAPE** together with some useful complex constants.

9.2.13.1 Complex Constants

```
CZERO = { 0., 0.}
CE = { 1., 0.}
CI = { 0., 1.}
CNE = {-1., 0.}
CNI = { 0., -1.}
CPI4 = {0.70710678, 0.70710678}
C3PI4 = {-0.70710678, 0.70710678}
C5PI4 = {-0.70710678, 0.70710678}
C7PI4 = {0.70710678, -0.70710678}
```

9.2.13.2 Complex Functions with one Complex Argument

`COMPLEX` **cneg** `(COMPLEX u)`
`-u`

`COMPLEX` **cinv** `(COMPLEX u)`
`1 / u`

`COMPLEX` **cconj** `(COMPLEX u)`
conjugate of `u`

`double` **creal** `(COMPLEX u)`
Re(u)

`double` **cimag** `(COMPLEX u)`
Im(u)

`double` **cnorm** `(COMPLEX u)`
`|u|`

`double` **carg** `(COMPLEX u)`
argument of `u`, return value is `-pi < carg(u) <= +pi`

`COMPLEX` **cexp** `(COMPLEX u)`
exp(u)

`COMPLEX` **csin** `(COMPLEX u)`
`sin(u)`

`COMPLEX` **ccos** `(COMPLEX u)`
`cos(u)`

`COMPLEX` **ctan** `(COMPLEX u)`
`tan(u)`

`COMPLEX` **ccot** `(COMPLEX u)`
`cot(u)`

`COMPLEX` **csinh** `(COMPLEX u)`
`sinh(u)`

`COMPLEX` **ccosh** `(COMPLEX u)`
`cosh(u)`

`COMPLEX` **ctanh** `(COMPLEX u)`
`tanh(u)`

`COMPLEX` **ccoth** `(COMPLEX u)`
`coth(u)`

`COMPLEX` **clog** `(COMPLEX u)`
logarithm of `u`, undefined on the negative x-axis

`COMPLEX` **csquare** `(COMPLEX u)`
`u * u`

`COMPLEX` **ccube** `(COMPLEX u)`
`u * u * u`

`COMPLEX` **csqrt** `(COMPLEX u)`
square root of `u`, undefined on the negative x-axis

9.2.13.3 Complex Functions with two Complex Arguments

`COMPLEX` **cadd** `(COMPLEX u, COMPLEX v)`
`u + v`

`COMPLEX` **csub** `(COMPLEX u, COMPLEX v)`
`u - v`

`COMPLEX` **cmul** `(COMPLEX u, COMPLEX v)`
`u * v`

`COMPLEX` **cdiv** `(COMPLEX u, COMPLEX v)`
`u / v`

`COMPLEX` **cpow** `(COMPLEX u, COMPLEX w)`
`u` to the power of `w`

9.2.13.4 Complex Functions with one Complex and one Real Argument

`COMPLEX` **cradd** `(COMPLEX u, double r)`
`u + r`

`COMPLEX` **crsub** `(COMPLEX u, double r)`
`u - r`

`COMPLEX` **crmul** `(COMPLEX u, double r)`
`r * u`

`COMPLEX` **crdiv** `(COMPLEX u, double r)`
`u / r`

`COMPLEX` **crpow** `(COMPLEX u, double r)`
`u` to the power of `r`

`COMPLEX` **casqrt** `(COMPLEX u, double r)`
square root of `u`, undefined on the line with angle `r`

`COMPLEX` **crot** `(COMPLEX u, double phi)`
rotation of `u` by the angle `phi`

9.2.13.5 Other Complex Functions

`COMPLEX` **cmplx** `(double x, double y)`
make a complex number with real part `x` and imaginary part `y`

`COMPLEX` **cpolar** `(double r, double phi)`
transformation from polar coordinates into complex numbers

`COMPLEX` **cmoebius** `(COMPLEX a, COMPLEX b, COMPLEX c, COMPLEX d, COMPLEX u)`
moebius transformation of the complex number `u`, `a - d` all complex

`COMPLEX` **crmoebius** `(double a, double b, double c, double d, COMPLEX u)`
moebius transformation of the complex number `u`, `a - d` all real

`COMPLEX` **cpower** `(COMPLEX u, int n)`
`u` to the power of `n` with integer `n`

`COMPLEX` **croot** `(COMPLEX u, int n)`
`n`th root of `u`, undefined on the negative x-axis

`COMPLEX` **cSKP** `(COMPLEX u[], COMPLEX v[], int n)`
`n`-dimensional complex scalar product

`COMPLEX` **cBIL** `(COMPLEX u[], COMPLEX v[], int n)`
multiplication of two `n`-dimensional complex vectors

`COMPLEX` **cSKP3** `(CVEC3 u, CVEC3 v)`
3-dimensional complex scalar product

`COMPLEX` **cBIL3** `(CVEC3 u, CVEC3 v)`
multiplication of two 3- dimensional complex vectors

Index

GRAPE Classes and Types

Adapt2d, 127
Amandus, 223
BND-FUNCTION, 127
Bnd_Curve, 109
Build_Parm, 215
Chain, 176
Cloud, 150
Colorbar, 170
Cons, 111
ControlDevice, 278
Curve_Parm, 209
Device, 267
Domain_Parm, 219
DVEC2, 105
DVEC3, 105
DVEC4, 105
Explicit, 219
Fe2d, 124
Fe3d, 134
G_List, 285
Geom2d, 130
GraphicDevice, 268
GraphicGt, 267
GraphicPs, 267
GraphicRay, 267
GraphicSoft, 268
GraphicVideo, 268
GraphicX11, 268
InOutFile, 281
InputFile, 281
INT3, 114
List_Of_Inst, 288
MATRIX44, 174
Mesh2d, 167
Mesh3d, 158
OutputFile, 281
REFINE2D, 127
Scene, 173
Streaklines, 151
Suprop, 174
SUPROP_DEV, 174
Surface, 189
Surface_Build, 215
TextDevice, 282
Time-Curve, 209
Time_Object, 183
Time_Parm, 184
TimeScene, 179
Timestep, 181
Trace, 149
Triang0d, 148
Triang1d, 105
Triang2d, 114
Triang3d, 134
VEC2, 105
VEC3, 105
VEC4, 105

GRAPE Methods Sorted by Classes

Adapt2d, coarsen-global-send, 128
Adapt2d, coarsen-local, 128
Adapt2d, deg-fine-disp, 128
Adapt2d, fit-bndfunctions-send, 129
Adapt2d, get-info, 129
Adapt2d, read, 128
Adapt2d, reduce-mem-send, 127
Adapt2d, refine-global-send, 128
Adapt2d, refine-local, 128
Adapt2d, set-rfc-field-send, 128
Adapt2d, write, 128
Amandus, assign-cons-send, 226
Amandus, close-send, 226
Amandus, compute-domain, 224
Amandus, compute-explicit, 224
Amandus, compute-gauss, 225
Amandus, compute-newton, 224
Amandus, compute-node, 224
Amandus, compute-send, 224
Amandus, compute-surface, 225
Amandus, delete, 223
Amandus, get-weier-f, 225
Amandus, list-alloc, 223
Amandus, list-free, 223
Amandus, merge-send, 223
Amandus, new-instance, 223
Amandus, newton-disp, 224
Amandus, open-in, 225

Amandus, read-data, 225
Amandus, reset-send, 226
Amandus, set-weier-f, 225
Bar1d, fix-min-max, 238
Bar1d, set-bar-color, 238
Bar1d, set-eval-fn, 238
Bar1d, set-min-max, 239
Bar1d, set-offset, 239
Bar1d, set-print-format, 239
Bar1d, set-scale, 239
Bar1d, set-step-size, 239
Bar1d, set-type, 239
Bar1d, set-value-changed, 239
Bar1d, set-var-and-type, 240
Bar1d, set-variable, 240
Bar1d, update, 240
Bnd_Curve, assign-curve-send, 111
Bnd_Curve, delete, 110
Bnd_Curve, display, 111
Bnd_Curve, hardcopy, 110
Bnd_Curve, interpol, 111
Bnd_Curve, interpol-fac, 111
Bnd_Curve, list-alloc, 110
Bnd_Curve, list-free, 110
Bnd_Curve, list-shrink-send, 110
Bnd_Curve, merge, 110
Bnd_Curve, new-instance, 109
Bnd_Curve, optimize-send, 110
Bnd_Curve, reflect, 111
Bnd_Curve, show-send, 111
Bnd_Curve, softcopy, 110
Build_Parm, delete, 215
Build_Parm, list-alloc, 216
Build_Parm, list-free, 216
Build_Parm, set, 216
Build_Parm, switch-send, 216
Button, default-action, 240
Button, new-instance, 240
Button, pop-layer-action, 240
Button, set-state, 241
Button, switch-state, 241
Chain, coarsen, 178
Chain, cons-close, 178
Chain, find-pick-index, 178
Chain, get-link, 177
Chain, get-link-post-obj, 178
Chain, get-link-pre-obj, 178
Chain, get-link-step, 177
Chain, get-object, 177
Chain, hardcopy, 177
Chain, interpol-fac, 178
Chain, merge, 177
Chain, merge-send, 177
Chain, softcopy, 177
Chain, universal, 178
Checkbox, new-instance, 242
Checkbox, set-checkfield-bits, 243
Checkbox, set-dependency-masks, 243
Checkbox, set-flag, 243
Checkbox, set-mask, 243
Checkbox, switch-flag, 243
Checkbox, switch-off, 243
Checkbox, switch-on, 243
CheckField, get-bitfield, 241
CheckField, get-free-mask, 241
CheckField, new-checkbox, 241
CheckField, new-instance, 242
CheckField, set-dependency-masks, 242
CheckField, set-mask-mode, 242
Cloud, display, 151
Cloud, interpol, 151
Color-Sel, new-instance, 244
Color-Sel, switch-color-model, 244
Colorbar, button, 172
Colorbar, change-values, 172
Colorbar, display, 171
Colorbar, get-stdcolorbar, 171
Colorbar, modify, 172
Colorbar, on-off, 172
Colorbar, set-default, 171
Colorbar, set-default-min-max, 172
Colorbar, set-default-values, 172
Cons, close-send, 113
Cons, delete, 112
Cons, display, 112
Cons, follow, 113
Cons, get-info, 113
Cons, get-triang, 113
Cons, hardcopy, 112
Cons, interpol, 113
Cons, interpol-fac, 113
Cons, list-alloc, 112
Cons, list-free, 112
Cons, list_shrink_send, 112
Cons, merge, 112
Cons, optimize-send, 112
Cons, reflect, 113
Cons, reset, 113
Cons, softcopy, 112
Cons, tube, 112
Cons, verify-use, 113
Curve-Parm, list-free, 209
Curve_Parm, delete, 209
Curve_Parm, list-alloc, 209
Cycle-Button, get-value, 244
Cycle-Button, new-instance, 244
Cycle-Button, set-value, 244
Domain_Parm, delete, 219

Domain_Parm, list-alloc, 220
Domain_Parm, list-free, 220
Domain_Parm, set, 220
Domain_Parm, switch-send, 220
Explicit, assign-cons-send, 221
Explicit, compute-explicit, 221
Explicit, compute-send, 221
Explicit, delete, 220
Explicit, list-alloc, 220
Explicit, list-free, 220
Explicit, merge-send, 220
Explicit, new-instance, 220
Explicit, open-in, 221
Explicit, read-data, 221
Explicit, set-compute-arg, 221
Explicit, switch-comp-send, 221
Fe2d, delete, 124
Fe2d, get-info, 126
Fe2d, interpol, 126
Fe2d, isoline, 124
Fe2d, isoline-disp, 124
Fe2d, print-value-send, 126
Fe2d, read, 126
Fe2d, rot, 126
Fe2d, set-dim-of-value, 126
Fe2d, vect, 125
Fe2d, vect-disp, 125
Fe2d, write, 126
Fe3d, add-grad-send, 143
Fe3d, bnd-iso-select-disp, 141
Fe3d, bnd-isoline-disp, 141
Fe3d, clip, 140
Fe3d, clip-in-out-disp, 140
Fe3d, clip-iso-select-disp, 140
Fe3d, clip-isoline-disp, 140
Fe3d, clip-vect-disp, 140
Fe3d, delete, 135
Fe3d, get-gradient, 143
Fe3d, level-disp, 142
Fe3d, make, 142
Fe3d, make6, 142
Fe3d, point-probe-send, 140
Fe3d, read, 135
Fe3d, write, 135
FileList, get-files, 245
FileList, new-instance, 245
Function-Bar1d, set-effect, 246
Function-Ruler, new-instance, 246
Function-Slider, new-instance, 246
G_List, append, 286
G_List, apply, 286
G_List, apply-method, 286
G_List, contains, 286
G_List, copy, 288
G_List, delete, 285
G_List, hardcopy, 288
G_List, is-empty, 286
G_List, new-instance, 285
G_List, softcopy, 288
Geom2d, assign-bnd-send, 133
Geom2d, bnd-disp, 132
Geom2d, close-dist-send, 133
Geom2d, close-send, 133
Geom2d, coarsen-global-send, 133
Geom2d, compute-normals-send, 132
Geom2d, copy-refine, 134
Geom2d, delete, 130
Geom2d, display, 131
Geom2d, get-bnd-index, 132
Geom2d, get-info, 132
Geom2d, get-vertex-wnormals-send, 133
Geom2d, hardcopy, 131
Geom2d, interpol, 132
Geom2d, interpol-fac, 132
Geom2d, list-alloc, 130
Geom2d, list-free, 130
Geom2d, list-shrink-send, 131
Geom2d, make-bnd-send, 133
Geom2d, merge, 131
Geom2d, new-instance, 130
Geom2d, normals, 131
Geom2d, normals-disp, 131
Geom2d, optimize-send, 130
Geom2d, point-list-alloc, 130
Geom2d, point-list-free, 130
Geom2d, refine-local-send, 133
Geom2d, reflect, 132
Geom2d, show-send, 132
Geom2d, skip-bnd-send, 133
Geom2d, softcopy, 131
Geom2d, sort-points-send, 131
GraphicDevice, copy, 273
GraphicDevice, get-pick-pos, 274
GraphicDevice, get-stddev, 267
GraphicDevice, grid-patch, 273
GraphicDevice, new-instance, 267
GraphicDevice, read, 273
GraphicDevice, set-grid-patch, 273
GraphicDevice, set-stddev, 267
GraphicDevice, write, 273
GraphicVideo, config-video, 278
GraphicVideo, delete, 277
GraphicVideo, new-instance, 277
GraphicVideo, start-video, 278
Group, add-inter, 247
Group, add-replacement, 247
Group, calc-inter, 247
Group, change-border, 247

Group, contains, 247
Group, count-entries, 247
Group, delete, 247
Group, get-entry, 248
Group, get-size, 248
Group, new-instance, 248
Group, remove-inter, 248
Group, remove-replacement, 248
Group, send-to-all-entries, 248
Group, set-redraw-flag, 248
Group, update, 248
identify-nbs-send, Triang3d, 139
Input, new-instance, 249
Input, set-print-format, 249
Input, set-tab-action, 249
Input, set-type, 249
Input, set-var-and-type, 249
Input, set-variable, 249
InputFile, delete, 281
InputFile, new-instance, 281
Interactive, add-layers, 250
Interactive, add-side-effect, 250
Interactive, default-action, 250
Interactive, is-active, 250
Interactive, layer-list-add, 250
Interactive, layer-list-contains, 250
Interactive, layer-list-remove, 250
Interactive, mark-side-effect, 251
Interactive, remove-layers, 251
Interactive, remove-side-effect, 251
Interactive, set, 251
Interactive, set-action, 251
Interactive, set-instance, 251
Interactive, set-method, 251
Interactive, set-redraw-flag, 251
Interactive, switch, 252
Item, get-group, 252
Item, is-visible, 252
Item, new-instance, 252
Item, set-add-action, 252
Item, set-color, 252
Item, set-color-index, 252
Item, set-fill-mode, 253
Item, set-label, 253
Item, set-pref-pos, 253
Item, set-pref-size, 253
Item, set-refresh, 253
Item, set-rel-size-mode, 253
Item, update, 253
Layer, hide, 254
Layer, is-visible, 255
Layer, new-instance, 255
Layer, remove, 255
Layer, set-fill-mode, 255
Layer, set-pref-pos, 255
Layer, show, 255
List_Of_Inst, activate_send, 289
List_Of_Inst, deactivate_send, 289
List_Of_Inst, delete, 288
List_Of_Inst, get_info, 290
List_Of_Inst, hardcopy, 289
List_Of_Inst, interpol, 290
List_Of_Inst, interpol_fac, 290
List_Of_Inst, list_alloc, 289
List_Of_Inst, list_free, 289
List_Of_Inst, merge, 289
List_Of_Inst, reflect, 289
List_Of_Inst, softcopy, 289
List_Of_Inst, universal, 289
Manager, add-inter, 256
Manager, add-inter-for-opt, 256
Manager, add-layer, 256
Manager, check-refresh, 256
Manager, delete-item, 256
Manager, disable-menu, 257
Manager, enable-all-layers, 257
Manager, enable-layer, 257
Manager, enable-menu, 257
Manager, get-current-object, 257
Manager, get-stdmgr, 257
Manager, handle, 258
Manager, new-handle, 258
Manager, pop-instance, 258
Manager, pop-layer, 258
Manager, pop-message, 258
Manager, process-events, 258
Manager, push-instance, 258
Manager, push-message, 259
Manager, recalc-cross, 259
Manager, remove-inter, 259
Manager, remove-layer, 259
Manager, remove-message, 259
Manager, send-to-layers, 259
Manager, set-clip-parm, 140
Manager, set-graphic-redraw, 259
Manager, set-message, 259
Manager, set-standard-message, 259
Manager, set-stdmgr, 260
Manager, update, 260
Mesh, clip, 159
Mesh, clip-ball-disp, 159
Mesh, clip-deg-fine-disp, 159
Mesh, clip-isoline-disp, 159
Mesh, clip-plane-disp, 159
Mesh, convert-to-mesh, 161
Mesh, display, 159
Mesh, level, 161
Mesh, level-disp, 161

Mesh, read-ascii-fdiff, 161
Mesh, read-binary-fdiff, 161
Mesh, shrink, 160
Mesh, shrink-disp, 160
Mesh, shrink-focus-element-disp, 160
Mesh, shrink-layer-disp, 160
Mesh, sub-level-disp, 161
Mesh, write-binary-fdiff, 161
Mesh2d, display, 169
Mesh2d, isoline, 169
Mesh2d, value-min-max, 169
Mesh2d, value-min-max-send, 169
MethList, get-methods, 260
MethList, new-instance, 260
Project, add, 94
Project, add-item, 94
Project, add-item-for-opt, 94
Project, delete, 91
Project, hide, 90
Project, new-instance, 90
Project, remove, 94
Project, set-object, 94
Project, setup, 91
Project, show, 90
Project, unuse, 94
Project, use, 93
Root, add-method, 59
Root, delete, 56, 60
Root, delete-method, 59
Root, find-class, 56
Root, find-method, 59
Root, follow, 70
Root, get-info, 69
Root, hardcopy, 64
Root, new-class, 56
Root, new-instance, 60
Root, print-classes-send, 69
Root, print-instances-send, 69
Root, print-methods-send, 69
Root, set-universal-send, 72
Root, show-send, 69
Root, softcopy, 64
Root, universal, 71
Root, unset-universal-send, 72
Ruler, new-instance, 261
Ruler, parameter-changed, 186
Scene, 2cloud-send, 151
Scene, continue-moving, 147
Scene, delete, 174
Scene, display, 174
Scene, expand, 175
Scene, find-pick, 175
Scene, first-moving, 147
Scene, fix-clip-send, 141
Scene, fix-level-send, 142
Scene, fix-sub-level-send, 142
Scene, fix-super-level-send, 142
Scene, generate-boundary-send, 139
Scene, interpol, 175
Scene, new-instance, 174
Scene, point-moving, 148
Scene, read, 176
Scene, refine-moving, 148
Scene, reflect-delete-send, 175
Scene, reflect-send, 175
Scene, split-surfaces-send, 122
Scene, store-suprop, 176
Scene, universal, 174
Scrollbar, new-instance, 261
Scrollbar, set-bar-size, 261
Scrollbar, set-total-size, 261
Slider, new-instance, 261
Spline-Editor, new-instance, 263
Spline-Editor, set-spline, 263
Spline-Editor, unset-spline, 263
Spline1d, assign, 262
Spline1d, eval, 262
Spline1d, get-min-max, 262
Spline1d, hardcopy, 262
Spline1d, inv-eval, 262
Spline1d, knots2bezier, 262
Spline1d, list-alloc, 262
Spline1d, list-free, 262
Spline1d, read, 262
Spline1d, set-linear, 263
Spline1d, write, 263
StaticText, new-instance, 263
StreakLine, display, 152
StreakLine, get-object, 152
StreakLine, get-times, 152
StreakLine, list-alloc, 152
StreakLine, list-free, 152
Suprop, load, 176
Suprop, read, 176
Suprop, store, 176
Suprop, write, 176
Surface, associate, 191
Surface, associate-disp, 191
Surface, close-send, 192
Surface, coarsen-send, 193
Surface, conjugate-disp, 190
Surface, debug, 69, 192
Surface, delete, 190
Surface, delete-tree-send, 192
Surface, display, 190
Surface, domain-disp, 190
Surface, find-pick, 192
Surface, follow, 192

Surface, gauss-disp, 190
Surface, geometry-disp, 190
Surface, get-asso, 193
Surface, get-bnd-curve, 191
Surface, get-info, 192
Surface, get-object, 191
Surface, hardcopy, 190
Surface, list-alloc, 190
Surface, list-free, 190
Surface, make-tree-send, 192
Surface, merge-send, 190
Surface, new-instance, 190
Surface, open-in, 191
Surface, read-data, 191
Surface, refine-all-send, 193
Surface, refine-send, 193
Surface, reflect, 193
Surface, reflect-delete-send, 193
Surface, reflect-send, 193
Surface, reset-send, 192
Surface, softcopy, 190
Surface, universal, 191
Surface, update, 193
Surface_Build, assign-cons, 217
Surface_Build, assign-cons-send, 217
Surface_Build, compute-send, 217
Surface_Build, compute-video-send, 217
Surface_Build, delete, 216
Surface_Build, list-alloc, 216
Surface_Build, list-free, 216
Surface_Build, merge-send, 216
Surface_Build, new-instance, 216
Surface_Build, open-in, 217
Surface_Build, read-data, 217
Surface_Build, ref-curv-disp, 217
Surface_Build, switch-comp-send, 218
TextDevice, get-error-device, 281
TextDevice, get-file-pointer, 281
TextDevice, get-stddev, 281
TextDevice, set-error-device, 281
TextDevice, set-stddev, 281
Textmessage, change-border, 264
Textmessage, get-text, 264
Textmessage, new-instance, 264
Textmessage, set-align, 264
Textmessage, set-text, 265
Time_Curve, compute-send, 210
Time_Curve, delete, 209
Time_Curve, get-info, 210
Time_Curve, get-object, 210
Time_Curve, list-alloc, 210
Time_Curve, list-free, 210
Time_Curve, make-tree-send, 211
Time_Curve, new-instance, 209
Time_Curve, open-in, 210
Time_Curve, read-data, 210
Time_Object, compute-send, 186
Time_Object, compute-video-send, 186
Time_Object, delete, 185
Time_Object, display, 186
Time_Object, follow, 187
Time_Object, geometry-disp, 186
Time_Object, get-frame-buttons, 187
Time_Object, get-info, 187
Time_Object, get-task-button, 188
Time_Object, get-time, 187
Time_Object, get-time-ruler, 187
Time_Object, hardcopy, 185
Time_Object, list-alloc, 185
Time_Object, list-free, 185
Time_Object, merge-send, 185
Time_Object, new-instance, 185
Time_Object, next-frame, 187
Time_Object, previous-frame, 187
Time_Object, read-data, 186
Time_Object, reset-send, 187
Time_Object, set-frame, 187
Time_Object, set-task, 188
Time_Object, set-time, 187
Time_Object, softcopy, 185
Time_Object, switch-comp-send, 188
Time_Object, universal, 187
Time_Object, update-frame, 187
Time_Object, update-time, 187
Time_Parm, delete, 184
Time_Parm, get-info, 184
Time_Parm, list-alloc, 184
Time_Parm, list-free, 184
Time_Parm, set, 185
Time_Parm, switch-send, 185
TimeScene, display, 179
TimeScene, follow, 180
TimeScene, get-info, 180
TimeScene, hardcopy, 179
TimeScene, read, 180
TimeScene, softcopy, 179
TimeScene, switch-sync-send, 180
TimeScene, synchronize, 180
TimeScene, universal, 179
TimeScene, write, 180
Timestep, assign-vec, 181
Timestep, delete, 181
Timestep, get, 182
Timestep, get-object, 182
Timestep, hardcopy, 181
Timestep, put, 182
Timestep, read, 181
Timestep, softcopy, 181

Timestep, tracing, 150
Timestep, tracing-disp, 150
Timestep, universal, 182
Timestep, write, 181
Trace, display, 149
Trace, read, 149
Trace, shorting, 149
Trace, tracing, 150
Trace, tracing-disp, 150
Trace, write, 149
Triang0d, display, 148
Triang1d, 2cloud, 151
Triang1d, c-show, 109
Triang1d, delete, 106
Triang1d, display, 107
Triang1d, get-info, 108
Triang1d, hardcopy, 106
Triang1d, interpol, 108
Triang1d, interpol-fac, 108
Triang1d, list-alloc, 106
Triang1d, list-free, 106
Triang1d, list-shrink-send, 106
Triang1d, merge, 106
Triang1d, optimize-send, 106
Triang1d, pipe, 107
Triang1d, points, 107
Triang1d, points-disp, 107
Triang1d, read, 108
Triang1d, reflect, 108
Triang1d, reset, 109
Triang1d, show-send, 108
Triang1d, softcopy, 106
Triang1d, tube, 107
Triang1d, tube-disp, 107
Triang1d, write, 108
Triang2d, 2cloud, 151
Triang2d, c-show, 119
Triang2d, center, 120
Triang2d, check-phase, 118
Triang2d, check-send, 118
Triang2d, clip, 116
Triang2d, clip-ball-disp, 116
Triang2d, clip-plane-disp, 116
Triang2d, delete, 115
Triang2d, display, 116
Triang2d, draw-normals-disp, 122
Triang2d, edges, 122
Triang2d, explicit, 121
Triang2d, find-pick, 120
Triang2d, find-pick-index, 120
Triang2d, find-pick-intersec, 120
Triang2d, fix-clip, 116
Triang2d, fix-clip-send, 116
Triang2d, get-info, 119
Triang2d, get-triangle-normals, 122
Triang2d, get-vertex-wnormals, 122
Triang2d, hardcopy, 116
Triang2d, identify-nbs-send, 121, 123
Triang2d, interpol, 119
Triang2d, interpol-fac, 119
Triang2d, inverse-orient-send, 122
Triang2d, list-alloc, 115
Triang2d, list-free, 115
Triang2d, list-shrink-send, 116
Triang2d, make-edge-conn, 123
Triang2d, make-triangle-conn, 123
Triang2d, mask-make, 120
Triang2d, merge, 116
Triang2d, mollify-send, 121
Triang2d, optimize-send, 116
Triang2d, orient-send, 122
Triang2d, point-list-alloc, 115
Triang2d, point-list-free, 115
Triang2d, points, 118
Triang2d, points-disp, 118
Triang2d, read, 118
Triang2d, refine4-send, 120
Triang2d, reflect, 119
Triang2d, regenerate-send, 123
Triang2d, reset, 119
Triang2d, rot, 121
Triang2d, show-edges-send, 122
Triang2d, show-send, 119
Triang2d, shrink, 117
Triang2d, shrink-angle-disp, 117
Triang2d, shrink-disp, 117
Triang2d, shrink-foc-point-disp, 117
Triang2d, shrink-foc-tria-disp, 117
Triang2d, shrink-layer-disp, 117
Triang2d, simple-rot, 121
Triang2d, softcopy, 116
Triang2d, sort-points-send, 122
Triang2d, write, 118
Triang3d, boundary-ident-send, 139
Triang3d, c-show, 135
Triang3d, center, 137
Triang3d, check-phase, 136
Triang3d, check-send, 136
Triang3d, clip, 137
Triang3d, clip-ball-disp, 137
Triang3d, clip-deg-fine-disp, 137
Triang3d, clip-plane-disp, 137
Triang3d, collect, 136
Triang3d, delete, 135
Triang3d, display, 135
Triang3d, find-pick-intersec, 137
Triang3d, fix-clip, 141
Triang3d, fix-level, 142

Triang3d, fix-sub-level, 142
Triang3d, fix-super-level, 142
Triang3d, generate-boundary, 139
Triang3d, identify-nbs-send, 139
Triang3d, level, 142
Triang3d, list-alloc, 136
Triang3d, list-free, 136
Triang3d, make, 138
Triang3d, make6, 138
Triang3d, mask-make, 138
Triang3d, mask-make6, 138
Triang3d, read, 135
Triang3d, select-boundary, 139
Triang3d, select-boundary-disp, 139
Triang3d, show-send, 135
Triang3d, shrink, 138
Triang3d, shrink-angle-disp, 138
Triang3d, shrink-disp, 138
Triang3d, shrink-focus-point-disp, 138
Triang3d, shrink-focus-tetra-disp, 138
Triang3d, shrink-layer-disp, 138
Triang3d, write, 135
XList, add-to-list, 265
XList, new-instance, 265
XList, set-sorting-mode, 265
XList-Button, new-for-filelist, 266
XList-Button, new-for-methlist, 266
XList-Button, new-for-xlist, 265

GRAPE Methods Sorted by Name

2cloud, Triang1d, 151
2cloud, Triang2d, 151
2cloud-send, Scene, 151
activate_send, List_Of_Inst, 289
add, Project, 94
add-grad-send, Fe3d, 143
add-inter, Group, 247
add-inter, Manager, 256
add-inter-for-opt, Manager, 256
add-item, Project, 94
add-item-for-opt, Project, 94
add-layer, Manager, 256
add-layers, Interactive, 250
add-method, Root, 59
add-replacement, Group, 247
add-side-effect, Interactive, 250
add-to-list, XList, 265
append, G_List, 286
apply, G_List, 286
apply-method, G_List, 286
assign, Spline1d, 262
assign-bnd-send, Geom2d, 133
assign-cons, Surface_Build, 217
assign-cons-send, Amandus, 226
assign-cons-send, Explicit, 221
assign-cons-send, Surface_Build, 217
assign-curve-send, Bnd_Curve, 111
assign-vec, Timestep, 181
associate, Surface, 191
associate-disp, Surface, 191
bnd-disp, Geom2d, 132
bnd-iso-select-disp, Fe3d, 141
bnd-isoline-disp, Fe3d, 141
boundary-ident-send, Triang3d, 139
button, Colorbar, 172
c-show, Triang1d, 109
c-show, Triang2d, 119
c-show, Triang3d, 135
calc-inter, Group, 247
center, Triang2d, 120
center, Triang3d, 137
change-border, Group, 247
change-border, Textmessage, 264
change-values, Colorbar, 172
check-phase, Triang2d, 118
check-phase, Triang3d, 136
check-refresh, Manager, 256
check-send, Triang2d, 118
check-send, Triang3d, 136
clip, Fe3d, 140
clip, Mesh, 159
clip, Triang2d, 116
clip, Triang3d, 137
clip-ball-disp, Mesh, 159
clip-ball-disp, Triang2d, 116
clip-ball-disp, Triang3d, 137
clip-deg-fine-disp, Triang3d, 137
clip-deg-fine-disp,Mesh, 159
clip-in-out-disp, Fe3d, 140
clip-iso-select-disp, Fe3d, 140
clip-isoline-disp, Fe3d, 140
clip-isoline-disp, Mesh, 159
clip-plane-disp, Mesh, 159
clip-plane-disp, Triang2d, 116
clip-plane-disp, Triang3d, 137
clip-vect-disp, Fe3d, 140
close-dist-send, Geom2d, 133
close-send, Amandus, 226
close-send, Cons, 113
close-send, Geom2d, 133
close-send, Surface, 192
coarsen, Chain, 178
coarsen-global-send, Adapt2d, 128
coarsen-global-send, Geom2d, 133
coarsen-local, Adapt2d, 128
coarsen-send, Surface, 193
collect, Triang3d, 136

compute-domain, Amandus, 224
compute-explicit, Amandus, 224
compute-explicit, Explicit, 221
compute-gauss, Amandus, 225
compute-newton, Amandus, 224
compute-node, Amandus, 224
compute-normals-send, Geom2d, 132
compute-send, Amandus, 224
compute-send, Explicit, 221
compute-send, Surface_Build, 217
compute-send, Time_Curve, 210
compute-send, Time_Object, 186
compute-surface, Amandus, 225
compute-video-send, Surface_Build, 217
compute-video-send, Time_Object, 186
config-video, GraphicVideo, 278
conjugate-disp, Surface, 190
cons-close, Chain, 178
contains, G_List, 286
contains, Group, 247
continue-moving, Scene, 147
convert-to-mesh, Mesh, 161
copy, G_List, 288
copy, GraphicDevice, 273
copy-refine, Geom2d, 134
count-entries, Group, 247
deactivate_send, List_Of_Inst, 289
debug, Surface, 69, 192
default-action, Button, 240
default-action, Interactive, 250
deg-fine-disp, Adapt2d, 128
delete, Amandus, 223
delete, Bnd_Curve, 110
delete, Build_Parm, 215
delete, Cons, 112
delete, Curve_Parm, 209
delete, Domain_Parm, 219
delete, Explicit, 220
delete, Fe2d, 124
delete, Fe3d, 135
delete, G_List, 285
delete, Geom2d, 130
delete, GraphicVideo, 277
delete, Group, 247
delete, InputFile, 281
delete, List_Of_Inst, 288
delete, Project, 91
delete, Root, 56, 60
delete, Scene, 174
delete, Surface, 190
delete, Surface_Build, 216
delete, Time_Curve, 209
delete, Time_Object, 185
delete, Time_Parm, 184
delete, Timestep, 181
delete, Triang1d, 106
delete, Triang2d, 115
delete, Triang3d, 135
delete-item, Manager, 256
delete-method, Root, 59
delete-tree-send, Surface, 192
disable-menu, Manager, 257
display, Bnd_Curve, 111
display, Cloud, 151
display, Colorbar, 171
display, Cons, 112
display, Geom2d, 131
display, Mesh, 159
display, Mesh2d, 169
display, Scene, 174
display, StreakLine, 152
display, Surface, 190
display, Time_Object, 186
display, TimeScene, 179
display, Trace, 149
display, Triang0d, 148
display, Triang1d, 107
display, Triang2d, 116
display, Triang3d, 135
domain-disp, Surface, 190
draw-normals-disp, Triang2d, 122
edges, Triang2d, 122
enable-all-layers, Manager, 257
enable-layer, Manager, 257
enable-menu, Manager, 257
eval, Spline1d, 262
expand, Scene, 175
explicit, Triang2d, 121
find-class, Root, 56
find-method, Root, 59
find-pick, Scene, 175
find-pick, Surface, 192
find-pick, Triang2d, 120
find-pick-index, Chain, 178
find-pick-index, Triang2d, 120
find-pick-intersec, Triang2d, 120
find-pick-intersec, Triang3d, 137
first-moving, Scene, 147
fit-bndfunctions-send, Adapt2d, 129
fix-clip, Triang2d, 116
fix-clip, Triang3d, 141
fix-clip-send, Scene, 141
fix-clip-send, Triang2d, 116
fix-level, Triang3d, 142
fix-level-send, Scene, 142
fix-min-max, Bar1d, 238
fix-sub-level, Triang3d, 142
fix-sub-level-send, Scene, 142

fix-super-level, Triang3d, 142
fix-super-level-send, Scene, 142
follow, Cons, 113
follow, Root, 70
follow, Surface, 192
follow, Time_Object, 187
follow, TimeScene, 180
gauss-disp, Surface, 190
generate-boundary, Triang3d, 139
generate-boundary-send, Scene, 139
geometry-disp, Surface, 190
geometry-disp, Time_Object, 186
get, Timestep, 182
get-asso, Surface, 193
get-bitfield, CheckField, 241
get-bnd-curve, Surface, 191
get-bnd-index, Geom2d, 132
get-current-object, Manager, 257
get-entry, Group, 248
get-error-device, TextDevice, 281
get-file-pointer, TextDevice, 281
get-files, FileList, 245
get-frame-buttons, Time_Object, 187
get-free-mask, CheckField, 241
get-gradient, Fe3d, 143
get-group, Item, 252
get-info, Adapt2d, 129
get-info, Cons, 113
get-info, Fe2d, 126
get-info, Geom2d, 132
get-info, Root, 69
get-info, Surface, 192
get-info, Time_Curve, 210
get-info, Time_Object, 187
get-info, Time_Parm, 184
get-info, TimeScene, 180
get-info, Triang1d, 108
get-info, Triang2d, 119
get-link, Chain, 177
get-link-post-obj, Chain, 178
get-link-pre-obj, Chain, 178
get-link-step, Chain, 177
get-methods, MethList, 260
get-min-max, Spline1d, 262
get-object, Chain, 177
get-object, StreakLine, 152
get-object, Surface, 191
get-object, Time_Curve, 210
get-object, Timestep, 182
get-pick-pos, GraphicDevice, 274
get-size, Group, 248
get-stdcolorbar, Colorbar, 171
get-stddev, GraphicDevice, 267
get-stddev, TextDevice, 281
get-stdmgr, Manager, 257
get-task-button, Time_Object, 188
get-text, Textmessage, 264
get-time, Time_Object, 187
get-time-ruler, Time_Object, 187
get-times, StreakLine, 152
get-triang, Cons, 113
get-triangle-normals, Triang2d, 122
get-value, Cycle-Button, 244
get-vertex-wnormals, Triang2d, 122
get-vertex-wnormals-send, Geom2d, 133
get-weier-f, Amandus, 225
get_info, List_Of_Inst, 290
grid-patch, GraphicDevice, 273
handle, Manager, 258
hardcopy, Bnd_Curve, 110
hardcopy, Chain, 177
hardcopy, Cons, 112
hardcopy, G_List, 288
hardcopy, Geom2d, 131
hardcopy, List_Of_Inst, 289
hardcopy, Root, 64
hardcopy, Spline1d, 262
hardcopy, Surface, 190
hardcopy, Time_Object, 185
hardcopy, TimeScene, 179
hardcopy, Timestep, 181
hardcopy, Triang1d, 106
hardcopy, Triang2d, 116
hide, Layer, 254
hide, Project, 90
identify-nbs-send, Triang2d, 121, 123
interpol, Bnd_Curve, 111
interpol, Cloud, 151
interpol, Cons, 113
interpol, Fe2d, 126
interpol, Geom2d, 132
interpol, List_Of_Inst, 290
interpol, Scene, 175
interpol, Triang1d, 108
interpol, Triang2d, 119
interpol-fac, Bnd_Curve, 111
interpol-fac, Chain, 178
interpol-fac, Cons, 113
interpol-fac, Geom2d, 132
interpol-fac, Triang1d, 108
interpol-fac, Triang2d, 119
interpol_fac, List_Of_Inst, 290
inv-eval, Spline1d, 262
inverse-orient-send , Triang2d, 122
is-active, Interactive, 250
is-empty, G_List, 286
is-visible, Item, 252
is-visible, Layer, 255

isoline, Fe2d, 124
isoline, Mesh2d, 169
isoline-disp, Fe2d, 124
knots2bezier, Spline1d, 262
layer-list-add, Interactive, 250
layer-list-contains, Interactive, 250
layer-list-remove, Interactive, 250
level, Mesh, 161
level, Triang3d, 142
level-disp, Fe3d, 142
leveldisp, Mesh, 161
list-alloc, Amandus, 223
list-alloc, Bnd_Curve, 110
list-alloc, Build_Parm, 216
list-alloc, Cons, 112
list-alloc, Curve_Parm, 209
list-alloc, Domain_Parm, 220
list-alloc, Explicit, 220
list-alloc, Geom2d, 130
list-alloc, Spline1d, 262
list-alloc, StreakLine, 152
list-alloc, Surface, 190
list-alloc, Surface_Build, 216
list-alloc, Time_Curve, 210
list-alloc, Time_Object, 185
list-alloc, Time_Parm, 184
list-alloc, Triang1d, 106
list-alloc, Triang2d, 115
list-alloc, Triang3d, 136
list-free, Amandus, 223
list-free, Bnd_Curve, 110
list-free, Build_Parm, 216
list-free, Cons, 112
list-free, Curve_Parm, 209
list-free, Domain_Parm, 220
list-free, Explicit, 220
list-free, Geom2d, 130
list-free, Spline1d, 262
list-free, StreakLine, 152
list-free, Surface, 190
list-free, Surface_Build, 216
list-free, Time_Curve, 210
list-free, Time_Object, 185
list-free, Time_Parm, 184
list-free, Triang1d, 106
list-free, Triang2d, 115
list-free, Triang3d, 136
list-shrink-send, Bnd_Curve, 110
list-shrink-send, Geom2d, 131
list-shrink-send, Triang1d, 106
list-shrink-send, Triang2d, 116
list_alloc, List_Of_Inst, 289
list_free, List_Of_Inst, 289
list_shrink_send, Cons, 112
load, Suprop, 176
make, Fe3d, 142
make, Triang3d, 138
make-bnd-send, Geom2d, 133
make-edge-conn, Triang2d, 123
make-tree-send, Surface, 192
make-tree-send, Time_Curve, 211
make-triangle-conn, Triang2d, 123
make6, Fe3d, 142
make6, Triang3d, 138
mark-side-effect, Interactive, 251
mask-make, Triang2d, 120
mask-make, Triang3d, 138
mask-make6, Triang3d, 138
merge, Bnd_Curve, 110
merge, Chain, 177
merge, Cons, 112
merge, Geom2d, 131
merge, List_Of_Inst, 289
merge, Triang1d, 106
merge, Triang2d, 116
merge-send, Amandus, 223
merge-send, Chain, 177
merge-send, Explicit, 220
merge-send, Surface, 190
merge-send, Surface_Build, 216
merge-send, Time_Object, 185
modify, Colorbar, 172
mollify-send, Triang2d, 121
new-checkbox, CheckField, 241
new-class, Root, 56
new-for-filelist, XList-Button, 266
new-for-methlist, XList-Button, 266
new-for-xlist, XList-Button, 265
new-handle, Manager, 258
new-instance, Amandus, 223
new-instance, Bnd_Curve, 109
new-instance, Button, 240
new-instance, Checkbox, 242
new-instance, CheckField, 242
new-instance, Color-Sel, 244
new-instance, Cycle-Button, 244
new-instance, Explicit, 220
new-instance, FileList, 245
new-instance, Function-Ruler, 246
new-instance, Function-Slider, 246
new-instance, G_List, 285
new-instance, Geom2d, 130
new-instance, GraphicDevice, 267
new-instance, GraphicVideo, 277
new-instance, Group, 248
new-instance, Input, 249
new-instance, InputFile, 281
new-instance, Item, 252

new-instance, Layer, 255
new-instance, MethList, 260
new-instance, Project, 90
new-instance, Root, 60
new-instance, Ruler, 261
new-instance, Scene, 174
new-instance, Scrollbar, 261
new-instance, Slider, 261
new-instance, Spline-Editor, 263
new-instance, StaticText, 263
new-instance, Surface, 190
new-instance, Surface_Build, 216
new-instance, Textmessage, 264
new-instance, Time_Curve, 209
new-instance, Time_Object, 185
new-instance, XList, 265
newton-disp, Amandus, 224
next-frame, Time_Object, 187
normals, Geom2d, 131
normals-disp, Geom2d, 131
on-off, Colorbar, 172
open-in, Amandus, 225
open-in, Explicit, 221
open-in, Surface, 191
open-in, Surface_Build, 217
open-in, Time_Curve, 210
optimize-send, Bnd_Curve, 110
optimize-send, Cons, 112
optimize-send, Geom2d, 130
optimize-send, Triang1d, 106
optimize-send, Triang2d, 116
orient-send, Triang2d, 122
parameter-changed, Ruler, 186
pipe, Triang1d, 107
point-list-alloc, Geom2d, 130
point-list-alloc, Triang2d, 115
point-list-free, Geom2d, 130
point-list-free, Triang2d, 115
point-moving, Scene, 148
point-probe-send, Fe3d, 140
points, Triang1d, 107
points, Triang2d, 118
points-disp, Triang1d, 107
points-disp, Triang2d, 118
pop-instance, Manager, 258
pop-layer, Manager, 258
pop-layer-action, Button, 240
pop-message, Manager, 258
previous-frame, Time_Object, 187
print-classes-send, Root, 69
print-instances-send, Root, 69
print-methods-send, Root, 69
print-value-send, Fe2d, 126
process-events, Manager, 258
push-instance, Manager, 258
push-message, Manager, 259
put, Timestep, 182
read, Adapt2d, 128
read, Class, 149, 180, 181
read, Fe2d, 126
read, Fe3d, 135
read, GraphicDevice, 273
read, Scene, 176
read, Spline1d, 262
read, Suprop, 176
read, TimeScene, 180
read, Timestep, 181
read, Trace, 149
read, Triang1d, 108
read, Triang2d, 118
read, Triang3d, 135
read-ascii-fdiff, Mesh, 161
read-binary-fdiff, Mesh, 161
read-data, Amandus, 225
read-data, Explicit, 221
read-data, Surface, 191
read-data, Surface_Build, 217
read-data, Time_Curve, 210
read-data, Time_Object, 186
recalc-cross, Manager, 259
reduce-mem-send, Adapt2d, 127
ref-curv-disp, Surface_Build, 217
refine-all-send, Surface, 193
refine-global-send, Adapt2d, 128
refine-local, Adapt2d, 128
refine-local-send, Geom2d, 133
refine-moving, Scene, 148
refine-send, Surface, 193
refine4-send, Triang2d, 120
reflect, Bnd_Curve, 111
reflect, Cons, 113
reflect, Geom2d, 132
reflect, List_Of_Inst, 289
reflect, Surface, 193
reflect, Triang1d, 108
reflect, Triang2d, 119
reflect-delete-send, Scene, 175
reflect-delete-send, Surface, 193
reflect-send, Scene, 175
reflect-send, Surface, 193
regenerate-send, Triang2d, 123
remove, Layer, 255
remove, Project, 94
remove-inter, Group, 248
remove-inter, Manager, 259
remove-layer, Manager, 259
remove-layers, Interactive, 251
remove-message, Manager, 259

remove-replacement, Group, 248
remove-side-effect, Interactive, 251
reset, Cons, 113
reset, Triang1d, 109
reset, Triang2d, 119
reset-send, Amandus, 226
reset-send, Surface, 192
reset-send, Time_Object, 187
rot, Fe2d, 126
rot, Triang2d, 121
select-boundary, Triang3d, 139
select-boundary-disp, Triang3d, 139
send-to-all-entries, Group, 248
send-to-layers, Manager, 259
set, Build_Parm, 216
set, Domain_Parm, 220
set, Interactive, 251
set, Time_Parm, 185
set-action, Interactive, 251
set-add-action, Item, 252
set-align, Textmessage, 264
set-bar-color, Bar1d, 238
set-bar-size, Scrollbar, 261
set-checkfield-bits, Checkbox, 243
set-clip-parm, Manager, 140
set-color, Item, 252
set-color-index, Item, 252
set-compute-arg, Explicit, 221
set-default, Colorbar, 171
set-default-min-max, Colorbar, 172
set-default-values, Colorbar, 172
set-dependency-masks, Checkbox, 243
set-dependency-masks, CheckField, 242
set-dim-of-value, Fe2d, 126
set-effect, Function-Bar1d, 246
set-error-device, TextDevice, 281
set-eval-fn, Bar1d, 238
set-fill-mode, Item, 253
set-fill-mode, Layer, 255
set-flag, Checkbox, 243
set-frame, Time_Object, 187
set-graphic-redraw, Manager, 259
set-grid-patch, GraphicDevice, 273
set-instance, Interactive, 251
set-label, Item, 253
set-linear, Spline1d, 263
set-mask, Checkbox, 243
set-mask-mode, CheckField, 242
set-message, Manager, 259
set-method, Interactive, 251
set-min-max, Bar1d, 239
set-object, Project, 94
set-offset, Bar1d, 239
set-pref-pos, Item, 253
set-pref-pos, Layer, 255
set-pref-size, Item, 253
set-print-format, Bar1d, 239
set-print-format, Input, 249
set-redraw-flag, Group, 248
set-redraw-flag, Interactive, 251
set-refresh, Item, 253
set-rel-size-mode, Item, 253
set-rfc-field-send, Adapt2d, 128
set-scale, Bar1d, 239
set-sorting-mode, XList, 265
set-spline, Spline-Editor, 263
set-standard-message, Manager, 259
set-state, Button, 241
set-stddev, GraphicDevice, 267
set-stddev, TextDevice, 281
set-stdmgr, Manager, 260
set-step-size, Bar1d, 239
set-tab-action, Input, 249
set-task, Time_Object, 188
set-text, Textmessage, 265
set-time, Time_Object, 187
set-total-size, Scrollbar, 261
set-type, Bar1d, 239
set-type, Input, 249
set-universal-send, Root, 72
set-value, Cycle-Button, 244
set-value-changed, Bar1d, 239
set-var-and-type, Bar1d, 240
set-var-and-type, Input, 249
set-variable, Bar1d, 240
set-variable, Input, 249
set-weier-f, Amandus, 225
setup, Project, 91
shorting, Trace, 149
show, Layer, 255
show, Project, 90
show-edges-send, Triang2d, 122
show-send, Bnd_Curve, 111
show-send, Geom2d, 132
show-send, Root, 69
show-send, Triang1d, 108
show-send, Triang2d, 119
show-send, Triang3d, 135
shrink, Mesh, 160
shrink, Triang2d, 117
shrink, Triang3d, 138
shrink-angle-disp, Triang2d, 117
shrink-angle-disp, Triang3d, 138
shrink-disp, Mesh, 160
shrink-disp, Triang2d, 117
shrink-disp, Triang3d, 138
shrink-foc-point-disp, Triang2d, 117
shrink-foc-tria-disp, Triang2d, 117

shrink-focus-element-disp, Mesh, 160
shrink-focus-point-disp, Triang3d, 138
shrink-focus-tetra-disp, Triang3d, 138
shrink-layer-disp, Mesh, 160
shrink-layer-disp, Triang2d, 117
shrink-layer-disp, Triang3d, 138
simple-rot, Triang2d, 121
skip-bnd-send, Geom2d, 133
softcopy, Bnd_Curve, 110
softcopy, Chain, 177
softcopy, Cons, 112
softcopy, G_List, 288
softcopy, Geom2d, 131
softcopy, List_Of_Inst, 289
softcopy, Root, 64
softcopy, Surface, 190
softcopy, Time_Object, 185
softcopy, TimeScene, 179
softcopy, Timestep, 181
softcopy, Triang1d, 106
softcopy, Triang2d, 116
sort-points-send, Geom2d, 131
sort-points-send, Triang2d, 122
split-surfaces-send, Scene, 122
start-video, GraphicVideo, 278
store, Suprop, 176
store-suprop, Scene, 176
sub-level-disp, Mesh, 161
switch, Interactive, 252
switch-color-model, Color-Sel, 244
switch-comp-send, Explicit, 221
switch-comp-send, Surface_Build, 218
switch-comp-send, Time_Object, 188
switch-flag, Checkbox, 243
switch-off, Checkbox, 243
switch-on, Checkbox, 243
switch-send, Build_Parm, 216
switch-send, Domain_Parm, 220
switch-send, Time_Parm, 185
switch-state, Button, 241
switch-sync-send, TimeScene, 180
synchronize, TimeScene, 180
tracing, Timestep, 150
tracing, Trace, 150
tracing-disp, Timestep, 150
tracing-disp, Trace, 150
tube, Cons, 112
tube, Triang1d, 107
tube-disp, Triang1d, 107
universal, Chain, 178
universal, List_Of_Inst, 289
universal, Root, 71
universal, Scene, 174
universal, Surface, 191
universal, Time_Object, 187
universal, TimeScene, 179
universal, Timestep, 182
unset-spline, Spline-Editor, 263
unset-universal-send, Root, 72
unuse, Project, 94
update, Bar1d, 240
update, Group, 248
update, Item, 253
update, Manager, 260
update, Surface, 193
update-frame, Time_Object, 187
update-time, Time_Object, 187
use, Project, 93
value-min-max, Mesh2d, 169
value-min-max-send, Mesh2d, 169
vect, Fe2d, 125
vect-disp, Fe2d, 125
verify-use, Cons, 113
write, Adapt2d, 128
write, Fe2d, 126
write, Fe3d, 135
write, GraphicDevice, 273
write, Spline1d, 263
write, Suprop, 176
write, TimeScene, 180
write, Timestep, 181
write, Trace, 149
write, Triang1d, 108
write, Triang2d, 118
write, Triang3d, 135
write-binary-fdiff, Mesh, 161