Programmer's Guide

For GNU MAVERIK version 6.1

Jon Cook, Toby Howard (Editors)

Roger Hubbold, Martin Keates, Simon Gibson,
Alan Murta, Steve Pettifer, Adrian West

Advanced Interfaces Group
Department of Computer Science
University of Manchester

June 14, 2001

Contents

Part I

Introduction

Chapter 1

Introduction

Welcome to the **MAVERIK Programmer's Guide** (MPG), which describes version 6.1 of GNU MAVERIK – the **MA**nchester **V**irtual **E**nvi**R**onment **I**nterface **K**ernel. In this manual we'll discuss the ideas behind MAVERIK, its architecture, the facilities it provides to application programmers, and also why we think it's novel and interesting. For complete documentation of the functions and types that make up the MAVERIK API, please refer the **MAVERIK Functional Specification** (MFS) [3] which is included in the MAVERIK distribution as postscript, pdf, HTML, and as on-line `man` pages. Generally, when a function is first mentioned in this manual, we cross-reference its main entry in the MFS.

Please also refer to the **MAVERIK Frequently Asked Questions** (FAQ) file, in the top-level directory of the MAVERIK distribution. For your convenience the FAQ is also listed in Appendix E (page 173), but the on-line version (`http://aig.cs.man.ac.uk/maverik/faq.htm`) is likely to be more up-to-date.

1.1 What is MAVERIK?

In its simplest form, MAVERIK is a C toolkit for managing display and interaction in stand-alone (that is, non-networked) single-user Virtual Environment applications. A complementary system under development, Deva [18, 21, 22, 8], provides a networked multi-user, multi-environment layer on top of MAVERIK, with the ability to efficiently specify behaviour, laws etc. As of release 4.3, MAVERIK is an official component of the Free Software Foundation's GNU Project located in Boston, USA (`http://www.gnu.org`). However, as the copyright holders of the original MAVERIK source we are able to distribute non-GPL'd versions of (our version of) MAVERIK under a commercial license. See `http://aig.cs.man.ac.uk/maverik/non-gpl.htm` for more details.

There are numerous other "VR toolkits" available, ranging from very low-level libraries of functions for drawing three-dimensional graphics and interacting with peripherals, to fully-blown "systems" that describe virtual environments in much higher level terms. MAVERIK lies somewhere in between these extremes. It provides an application with the tools needed to create, manage, view, interact with,

and navigate around graphically complex Virtual Environments while making the minimum number of assumptions about the nature of the application.

MAVERIK does not dictate the use of any fixed object/scene representations or viewing/interaction techniques. Rather, it has the ability, where needed, to directly link into and exploit an application's own data structures and algorithms. This novel aspect of MAVERIK allows it to easily take advantage of representations, optimisations, and techniques that are highly application specific giving the resulting virtual environment a behaviour which is customized to, and consistent with, the nature of the application.

MAVERIK's flexible design means that applications with widely differing requirements can be supported.

MAVERIK has two components:

- a **micro-kernel**, which provides the framework within which applications are built;
- a collection of **supporting modules**, which provide optimised display management, culling, spatial management, interaction and navigation techniques, control of input and output devices etc. These modules are distributed as source code and act as a basis for customization.

It is important to appreciate that MAVERIK is not an "end-user application": there are no graphical user interfaces or "world editors" – it is strictly a programming tool.

A more detailed description of MAVERIK's architecture and design philosophy is given in the next chapter.

1.2 What platforms does MAVERIK support?

MAVERIK is available as source code and should compile under Windows, MacOS and on UNIX systems – essentially any system that has OpenGL, Mesa (version 3.1 or above), IrisGL or DirectX (version 7). However, while it is possible to use any of these libraries, OpenGL/Mesa is currently the best supported library for MAVERIK to use.

MAVERIK is known to run on the following operating systems:

- SGI Irix 5.3, 6.3 and 6.5;
- RedHat 5.2 and 6.x;
- FreeBSD 3.2;
- SuSE 7.1;
- SunOS 5.7;

- Windows 98, 2k and NT;
- MacOS;

This list is not intended to be exhaustive but simply reflects operating systems that we, or others, have access to and tried MAVERIK with. Ports to other UNIX platforms should be fairly trivial and we believe the code to work on Window 95.

Since MAVERIK uses well supported graphics libraries to perform rendering (OpenGL, IrisGL or DirectX) it can take advantage of the hardware acceleration available on certain graphics cards. For example, as well as our SGI's, we use MAVERIK on PCs, running GNU/Linux which are equipped with GeForce2 graphics cards (we also use a machine fitted with two Voodoo2 cards in order to produce stereo output).

1.3 What peripherals does MAVERIK support?

A standard compilation of MAVERIK provides supports for a desktop mouse, keyboard and screen. This makes it easy to try out the examples and demonstrations.

The configuration of 3D peripherals used in VR labs tends to be site specific. Code is included in the distribution to support Polhemus FASTRAK and ISOTRAK II six degree of freedom trackers (optionally coupled to Division 3D mice); Ascension Flock of birds (ERC only); Spacetec SpaceBalls and SpaceOrb360s; Magellan Space Mouse; InterSense InterTrax 30 gyroscopic trackers; 5DT data gloves; and a serial Logitech Marble Mouse. With modification other similar specification 6 DOF input devices/tracking technology can be supported. This code is not compiled by default since it is not relevant to everyone and requires some manual configuration. See the `README` in the `src/extras` sub-directory of the MAVERIK distribution for more information.

We have also supported more peculiar peripherals in our own lab: IBM ViaVoice speech recognition system; Microsoft SideWinder Force-Feedback joystick and our homebuilt MIDI server. These are relatively uncommon devices and so are not included in a "standard" MAVERIK release. If you're interested in this code, please contact us.

1.4 What has MAVERIK been used for so far?

The development of MAVERIK began in 1997, since when it has been used for many different projects and applications, including:

- research into the improvement of interfaces to complex engineering tasks, such as the design and operation of off-shore drilling platforms [13, 12, 24, 4, 10];
- large-scale electronic landscapes for way-finding and public information access [19, 14, 23, 20];

- stereoscopic modelling of scenes of crime [9, 15];
- abstract data visualisation [17, 16];
- visualisation of physically based simulations [7, 5, 6];
- electronic artworks [2];
- tools for interactively creating and editing virtual environments;
- modelling nanotechnology;
- architectural modelling.

You can find details of these and other projects at the Advanced Interfaces Group's MAVERIK applications Web page – `http://aig.cs.man.ac.uk/gallery/index.htm`.

1.5 MAVERIK levels

The MAVERIK API comprises over 550 functions, only a small subset of which will commonly be used by programmers wishing to use MAVERIK "out of the box". Similarly, many functions will be of interest only to those users wishing to understand the internal workings of MAVERIK, and possibly wishing to tailor it to their own requirements.

With these various requirements in mind, we have divided the MAVERIK functionality into three "levels", which we hope will help users to find their way around. This three-level structure is reflected both in this manual, and in the MAVERIK Functional Specification.

- **Level 1** functions are those which first-time users of MAVERIK will normally use. These functions make use of the many defaults built into MAVERIK, and should enable users to create interesting MAVERIK applications quickly.
- **Level 2** functions are those which allow more advanced use of MAVERIK. Examples might include defining new classes of object, or defining new methods of navigating around the virtual environment.
- **Level 3** functions are intended for "Research and Development" using MAVERIK. They are low-level functions which provide interfaces to the MAVERIK kernel and associated modules. For example, Level 3 functions would be required for extending MAVERIK to provide new level-of-detail processing algorithms, new object culling algorithms, or to provide support for new kinds of input devices.

1.6 Assumed readers' background

Because MAVERIK is a research and development system, we assume that the reader is already familiar with the basic concepts of computer graphics and virtual environments. In particular, we assume

that the reader is comfortable with the ideas of modelling coordinates and world coordinates; transformations; rendering in the OpenGL style; callback functions; and the object-oriented programming ideas of classes and methods.

1.7 Contact

Comments, questions and feedback are actively encouraged and should be addressed to us personally at `maverik@aig.cs.man.ac.uk`, or to the MAVERIK user's mailing list (details of which can be found at `http://aig.cs.man.ac.uk/contact.htm`). Bugs should be reported to the mailing list, or to `bug-maverik@aig.cs.man.ac.uk`, but only after you have consulted the `FAQ` and list of known bugs.

1.8 Authors and contributors

The following people are responsible for the design, development and implementation of MAVERIK (in alphabetical order): Jon Cook, Tim Davis, Simon Gibson, Toby Howard, Roger Hubbold, Martin Keates, Alan Murta, Steve Pettifer, Adrian West. We are also indebted to the many valuable contributions of the following research students: Mat Brooks, Mashhuda Glencross, James Marsh, Gary Ng, Dan Oram, James Pearce, James Sinnott and Dongbo Xiao.

We would also like to thank the following people for contributing to MAVERIK: Robert Belleman (Solaris support), Shamus Smith (ISOTRAK II support), Joerg Anders (Windows support), Alex (MacOS support), Joe Topjian (FreeBSD support), Jake Burkholder (FreeBSD support), Rob G (FreeBSD support); Daniel Amos (Ascension Flock of Birds support); Alessandro De Luca (SpaceOrb360 and 5DT data glove support); and to everyone who mailed us with bug reports and fixes.

1.9 Acknowledgements

It's a pleasure to have the opportunity to acknowledge the organisations and individuals who have made the development of MAVERIK possible.

We thank the UK Engineering and Physical Sciences Research Council (EPSRC) for funding the VRLSA (GR/K99701) and REVEAL (GR/M14531) projects; the ESPRIT programme for funding eSCAPE (ESPRIT 25377); our academic research partners at the Universities of Manchester, Lancaster and Nottingham, the Swedish Institute of Computer Science, and ZKM in Karlsruhe; our industrial partners CADCentre Ltd, Sharp Laboratories of Europe Ltd, Brown & Root, Greater Manchester Police and Harlequin Ltd.

Chapter 2

MAVERIK's architecture and design

This chapter briefly describes the architecture and design principles behind MAVERIK. For a more detailed description please see [11].

MAVERIK was designed to be a Virtual Reality system which addresses two key concerns: easy **customisation** to meet the demands of different applications, and **efficient operation** so that very large environments can be handled. Our approach adopts a "micro-kernel" design which minimises assumptions about how environments are represented and stored by the system.

MAVERIK is one of two components in a complete VR "operating system" under development within the Advanced Interfaces Group. We refer to MAVERIK as a micro-kernel because it provides a core set of functions for implementing VR interfaces on behalf of a single user. The second component, called Deva, provides a higher-level operating environment supporting multiple users, distributed shared environments, and multiple persistent concurrent environments. We do not discuss Deva further in this manual; for details please see [18, 8].

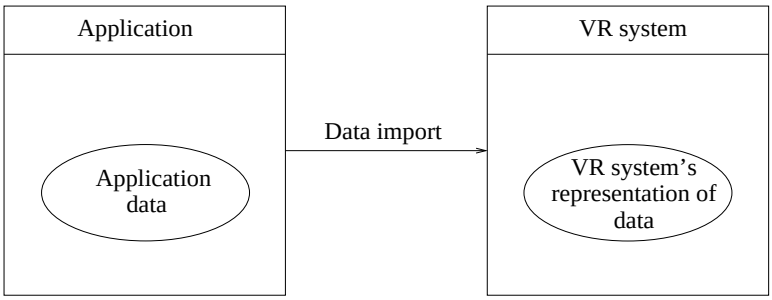
This chapter presents a description of the MAVERIK micro-kernel, its features and architecture, and how it compares to other VR software systems.

2.1 Virtual environment representations

One defining characteristic of a VR system is the way in which representations of virtual environments are stored and manipulated internally. In this section we contrast two common approaches: **fixed representations** and **immediate-mode rendering**.

2.1.1 Fixed representations

Most VR systems use a **fixed representation** for the virtual environment. This is shown in the figure below:



Here, an application's data is imported into a separate VR system, which stores it internally in some fixed format optimised for its own use. Often this will be a polygonal representation of the geometry of the model in a format dictated by the underlying graphics system.

This approach works well if the required VE has limited possibilities for interaction, such as a simple walk-through, or interaction with only a small number of objects. The advantage of the approach is that a large number of "standard" capabilities suited to the chosen representation – for example, culling and interaction mechanisms – can be provided as part of the VR system's intrinsic design and implementation.

However, as the complexity of an environment increases, and the requirement arises to associate appropriate behaviours and affordances with objects, the "fixed representation" approach become problematic. The data required to achieve such behaviour will always be application specific – since only the application can "know" what an object actually is, and what it means in the VE. Finding sensible mappings between a semantically rich application database and the restricted graphics-oriented data structure of the VR system is usually difficult and often unsatisfactory. Furthermore, it is often very difficult to exploit this information to affect the behaviour of the core VR system's functionality – for example, its culling, level-of-detail and navigation routines – since the user has little or no access to these.

Choosing a common representation, suitable for widely differing applications, is a difficult task, inevitably involving a trade-off between conflicting interests. For example, the needs of an application involved in design work for the motor industry are clearly quite different from those involved in abstract data visualization. A consequence of application diversity is that with a fixed representation, it is difficult to create a truly general-purpose VR system which can exploit application semantics.

The "fixed representation" scheme has another drawback: it requires that the two separate representations of the same underlying data must be maintained, one for the application and one for the VR system. It is a non-trivial programming task to keep separate representations synchronized.

2.1.2 Immediate-mode rendering

An alternative to storing graphical data in a separate fixed data structure as described above is to use immediate-mode rendering. Here, pictures are generated algorithmically, directly from an application's data, by writing a program in a language such as C. Calls to functions in a graphics library, such as OpenGL, embedded within the application, send data directly to the graphics hardware for

immediate rendering.

The big advantage of this approach is its ability to directly use arbitrary application data structures and also to exploit the application's algorithms to give the VE meaningful behaviour. The disadvantage is that since no standard representation is used, the graphics library cannot provide **higher level functionality**, such as culling and navigation.

2.2 The MAVERIK system

Like OpenGL, MAVERIK can be thought of as a graphics library which links into an application and directly uses its data structures and algorithms.

The crucial difference is that it also defines a standardized framework in which an application provides MAVERIK with the means to access its objects. Through the use of this framework, MAVERIK can provide high level functionality without dictating the use of any specific object representation.

MAVERIK has an object-oriented structure. It defines a set of **classes** for different kinds of object, and mechanisms for defining new classes. Customisation for different applications is achieved by defining **methods** associated with each class.

MAVERIK is implemented in standard C, so that it can be ported easily to different platforms and can be used by anyone with basic C programming skills. Methods are implemented using **callback functions**, with data passed via generic "typeless" pointer parameters. Note, however, that class hierarchies and inheritance are not supported.

2.2.1 Object definition

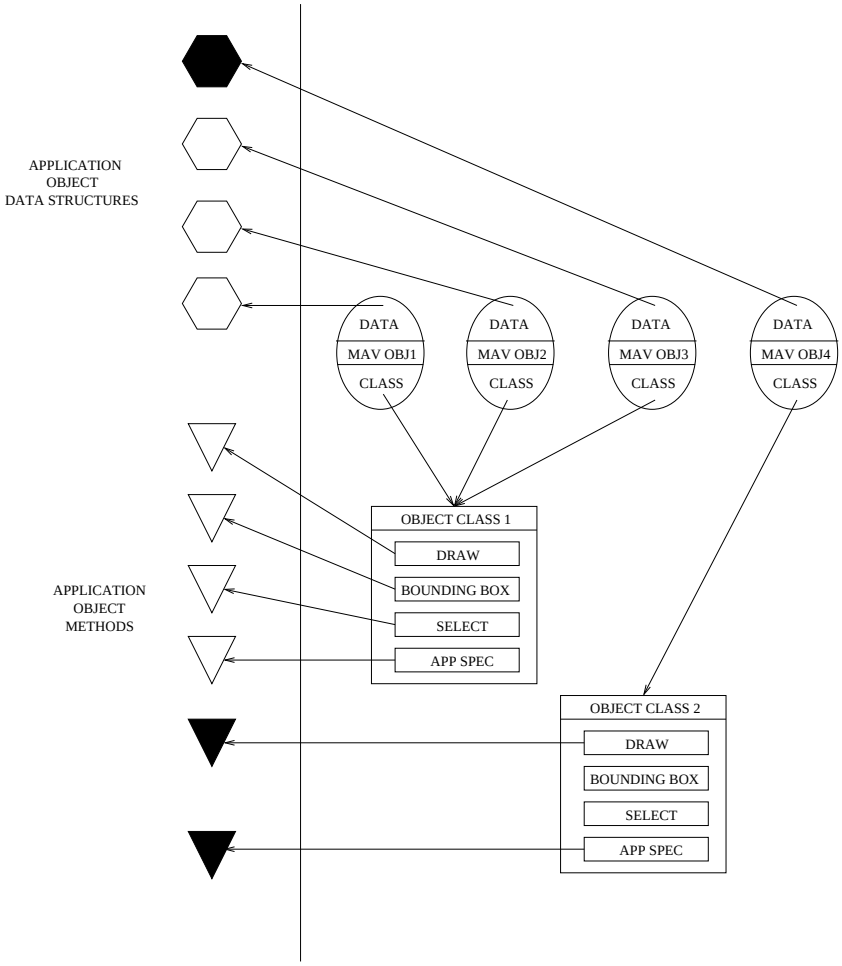
An "object" is simply a convenient way of naming something which an application requires MAVERIK to treat as an entity. No assumptions are made about how an object is represented by the application. For example, an object might be a single polygon, a group of polygons defining some more complex shape, such as a desk or chair, or some group of more complex primitive shapes which are specific to that one application – such as a ladder, or a valve.

The way to define different kinds of objects is to create a **class** for each one. This is done by calling a function, which returns a unique identifier for the new class. Different classes each have a (possibly unique) set of methods, implemented as C functions accessed as callbacks. Methods govern operations such as displaying primitives, computing their bounding boxes, or finding objects which are spatially closest to a given point. MAVERIK arranges that these methods are called to render frames. Methods which are specific to a particular application can also be defined, such as computing the mass of an object, or finding its centre of mass. Generally speaking, the minimum set of methods necessary to create a simple interactive VE comprises those for displaying objects, for computing a bounding volume, and for selecting/manipulating them (usually by 'grasping' or pointing at them in some way).

To avoid the tedium of having to write callback functions every time a new application is implemented,

MAVERIK provides **default methods** for a few common primitives such as polygons, polygon meshes, spheres, cylinders, cones, tori, boxes, and sub-parts of these (such as an angular section of a cylinder or torus). These default methods are distributed as source code, providing a set of examples and facilitating customisation.

As well as defining classes and associated methods, individual objects to be managed by MAVERIK must be **registered**. This is performed by a function which takes as input an object's class and a pointer to the data defining that object. This function binds these two elements into a single MAVERIK object, whose identifier is returned for use in subsequent references. In this way, objects are stored so that MAVERIK can find the class of any registered object – and hence any associated methods – and can also pass to the callback functions the generic pointer to the application data. Callback functions perform a cast into a pointer of the correct type for the data. The figure below illustrates the MAVERIK framework for objects, classes and methods:



The application's data structures are shown (as hexagons) to the left of the vertical line. The application's methods, which act upon the objects, are drawn as triangles. The shading illustrates which algorithms operate on which data structures (black on black, white on white).

The framework by which MAVERIK can access the application is shown on the right of the vertical

line. Each MAVERIK class contains a number of callback functions to process each class of object. The MAVERIK **objects** are the encapsulation of the appropriate class and the application-specific data structures which define the object.

Because MAVERIK objects only maintain **pointers** to – not **copies** of – the data structure and class, they do not have to be notified of any changes to them.

Although not shown in the Figure, MAVERIK uses a similar callback mechanism for registering event handlers and navigation functions.

The standard distribution of MAVERIK contains libraries of default methods for displaying and managing many common types of graphical primitives, and for navigation around the virtual environment. These can be customised easily by adding extra data and code, or simply replaced by alternative versions which are intimately bound to the data structures used by the application.

2.2.2 Spatial management structures

Another important feature of a VR system is its support for **spatial management** – this is central to many algorithms and techniques, such as culling, object selection, and collision detection, and is essential for managing large models. A common approach is to use a hierarchy of bounding volumes for spatial searching, which generally works efficiently because of its logarithmic complexity. However, as with object storage, it is possible to find optimisations which capitalise on application-specific features to yield superior performance.

MAVERIK provides a framework which permits customisation of spatial management methods. In a manner analogous to object definition, MAVERIK uses classes and methods to store and access spatial data. An application defines a class for each object storage technique, registers the callback functions corresponding to the different methods for each class, and defines generic object management structures – called **spatial management structures** (SMSs) – to store and manage MAVERIK objects.

Typical methods associated with SMSs include **object insertion**, **object deletion** and **cull to a region of space**. However, as with objects, an application can define whatever new classes and associated methods are most appropriate. An example of application-specific SMS processing might be to enforce a minimum spatial separation between objects.

As with objects, default methods are supplied which implement a range of useful techniques. One default class of SMS stores objects as a simple linked list, and processes them (for example, for display) in the order in which they were inserted, but only if an object's bounding box lies inside the current view frustum. Another class of SMS implements a hierarchy of bounding volumes. Any application-specific object that provides the "calculate bounding box" method can be used with these spatial management structures.

Although SMSs, as their name implies, are generally used for spatial management, objects can be stored in a non-spatial manner. For example, a linear list which maintains objects in insertion order is usually non-spatial. Such a list can be re-ordered to optimise graphics hardware context changes during display. Alternatively, objects could be sorted on a particular application-specific data field in order to accelerate processing of other kinds of queries.

Multiple SMSs

Objects can be inserted in any number of SMS's and processing can be performed on the SMS most suited to a particular task. One case where a simple linear list is useful is object manipulation. Suppose that a hierarchy of bounding volumes (HBV) is the default SMS for a large-scale model. Objects to be manipulated can be temporarily removed from the HBV SMS and inserted into a simple linked list for the duration of the manipulation. Subsequently, they can be reinserted into the HBV structure. The advantage of this is that potentially expensive alterations to the HBV structure are not needed during dynamic changes to the model. Because MAVERIK can manage several SMS structures simultaneously, the programming effort required to manage this is small.

A second example of multiple SMSs is to use one for view frustum culling and a second for object display. The first structure is used to flag visible objects and is organised for efficient spatial searching. A good choice for this would be an HBV. Objects referenced in the HBV are actually stored in the second SMS, which is ordered to minimise graphics context switches. This second SMS is then traversed displaying only the flagged visible objects.

Data consistency is maintained because all SMSs store **references** to MAVERIK objects, which in turn contain references to the application-specific objects. MAVERIK maintains, for each object, a list of the SMSs into which it has been inserted, and automatically removes it from each SMS if the object is deleted.

2.3 Summary

The design approach we have adopted for MAVERIK has three advantages:

- First, none of the application data is imported into, or replicated within, MAVERIK. This avoids the problem of synchronising changes to multiple representations.
- Second, the framework encapsulates all the information needed by MAVERIK to access data and methods stored externally within the application, so that object classes can be reused easily in other applications.
- Third, the philosophy is simple to understand and use, and straightforward to link to existing applications. This last point is important in domains such as CAD, where there is a major legacy problem with large-scale databases and existing code.

It might be argued that other VR systems can be tailored in much the same way. For us, the issue is the ease with which alternative behaviours can be implemented. The callback mechanism is a familiar programming technique, popularised by windowing systems such as X Windows, and graphics systems such as OpenGL. In MAVERIK a simpler parameter-passing mechanism has been adopted than that in X. Our design provides a clean interface which enables customisation to be configured dynamically at run-time. Callbacks can be switched (re-registered) to change the dynamic behaviour of the system.

For example, suppose that an environment comprises a city populated by buildings which the user is permitted to enter and move around. The insides of the buildings and the city outside may be optimised to use completely different spatial management methods for culling, navigation and interaction. Specifically, we use an occlusion culling algorithm for displaying the city, and a cell and portal method for the interior of the buildings. In MAVERIK, we can treat the objects representing the exterior of the buildings as belonging to a different class from those on the insides, and we register appropriate methods for each. However, the method employed by the user for moving around may need to be changed dynamically at run-time. Thus, the methods for a user walking around the streets or inside a building will be quite different from one driving a virtual car – the constraints and affordances in each case will be quite different. The navigation methods can be re-registered as the user enters or leaves buildings, or climbs into the car.

The remainder of this manual describes MAVERIK from a programmer's point of view. In the next chapter we present an introduction to programming with MAVERIK, using a series of worked example programs, all of which are available in the MAVERIK distribution.

Part II

MAVERIK Programming Level 1

Chapter 3

Introduction to MAVERIK programming

In this chapter we introduce some of the fundamental MAVERIK concepts in enough detail to allow you to write simple applications.

We'll present and work through a set of example programs, each of which builds on the previous example, as follows:

- Example 1: a minimal MAVERIK application;
- Example 2: defining and displaying an object;
- Example 3: surface parameters and navigation around the virtual environment;
- Example 4: a more complex environment with multiple objects.

The source code for the example programs in this manual, along with the `Makefile` to build them, can be found in the `examples/MPG` directory of the MAVERIK distribution. If you installed MAVERIK yourself, you'll know where this is. If not, ask your friendly system administrator.

We suggest you take copies of the examples, and the `Makefile`, to familiarise yourself with compiling and linking with MAVERIK. See Appendix A (page 125) for full details of how to compile and execute these examples.

3.1 The structure of a MAVERIK application

Broadly speaking, a MAVERIK application has a simple logical structure, comprising the following five sections:

- MAVERIK initialisation;

- Define the objects which comprise the virtual environment;
- Define the application's "behaviour" – how objects are managed, responds to interactions, defining navigation etc;
- Enter the MAVERIK rendering and interaction loop; once entered, this loop never quits, until the application does;
- Within each cycle of the loop, react to interaction events, and draw a frame.

3.2 Example 1: A minimal MAVERIK application

Our first example program is about as minimal as it's possible to get, but we hope it will serve to illustrate the logical structure outlined above. It will also ensure that you have correctly compiled the example, have a working version of the MAVERIK library, and that any paths are correctly set.

Try compiling and running `eg1.c`. You should see a window appear with the MAVERIK welcome message in it. This message consists of a spiraling MAVERIK logo with various copyright, version and contact information displayed. By default this message appears at the start of every MAVERIK application.

When the message clears you should see an empty blue window. The window will sit there forever, or until you move the mouse focus into the window, and type **Shift-Esc** on the keyboard. This key sequence is recognised by all MAVERIK applications and causes them to quit.

Here's the source code for `eg1.c`:

```
/* eg1.c */
#include "maverik.h"

int main(int argc, char *argv[])
{
  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  /* Rendering loop */
  while (1) {

    /* Check for and act on any events */
    mav_eventsCheck();

    /* Request start of a new frame */
    mav_frameBegin();

    /* Request end of the frame */
    mav_frameEnd();
  }
}
```

The program begins with an include file. `maverik.h` is the standard MAVERIK include which must appear in all MAVERIK programs. It contains all the definitions for MAVERIK constants, typedefs, and prototypes for the MAVERIK functions.

MAVERIK must be initialised before it can be used. Either one of two functions can be used to perform this – `mav_initialise` (MFS p 144) or `mav_initialiseNoArgs` (MFS p 144) – and one of these must be the first MAVERIK function called by the application. The two function are essentially the same, the difference being the former takes the command line arguments which can be used to control the initialisation process (see Appendix D (page 169) for a full description of this).

By default initialisation opens a screen window for rendering which will be a quarter of the overall screen size, and positioned in the lower left quadrant of the screen.

The shape and position of the window created by the initialisation call are examples of a number of aspects of MAVERIK's behaviour which are controlled by a set of global variables. These global variables are named `mav_opt_*`, and their default values can be explicitly overwritten by an application. For example, setting the variables `mav_opt_x`, `mav_opt_y`, `mav_opt_width` and `mav_opt_height` prior to the initialisation call allows an application to customise the position (bottom-left) and size of the window opened by `mav_initialise`. See Appendix C (page 155) for a complete list of the `mav_opt_*` variables.

Once initialised, MAVERIK is ready for use. In this example, we immediately enter the main rendering and interaction loop without defining any objects or "behaviour". The main loop typically has the following structure:

- The application calls `mav_eventsCheck` (MFS p 136) to check if any interaction events have occurred (triggered, for example, by the use of a mouse or keyboard). If MAVERIK detects that any events have occurred, it automatically calls functions to process the events. We'll describe how this works in Chapter 4 (page 35). Calling `mav_eventsCheck` also triggers navigation, as we'll see in Example 3.
- Next, the application calls `mav_frameBegin` (MFS p 139) to request MAVERIK to start a new rendering frame. `mav_frameBegin` actually causes quite a few things to happen behind the scenes, which we'll discuss later. MAVERIK uses double-buffering by default, so for now, think of this function as just clearing the back buffer in preparation for rendering a new frame.
- The next step would be to ask MAVERIK to do something useful for us, which would normally be to request an up-to-date display of all the objects in the virtual environment. We'll discuss this in Example 2.
- Finally, we call `mav_frameEnd` (MFS p 141) to inform MAVERIK that the frame is now complete, and ready for display. MAVERIK then swaps the buffers and updates the display (assuming we are using the default double-buffered configuration).

  The wallclock time elapsed between the calls to `mav_frameBegin` and `mav_frameEnd` gives the time taken to render a frame. The reciprocal of this value, the frame-rate, is stored in the global MAVERIK variable `mav_fps`, which the application can consult.

  For example, you could print it in the shell window using the following code (placed after `mav_frameEnd`):

```
printf ("frame rate: %4.2f\n", mav_fps);
```

N.B. For high frame rates (short elapsed time) this value will inevitably fluctuate from frame to frame due to variations in system load and the resolution and inaccuracies of the internal clock. The variable `mav_fps_avg` gives the frame rate averaged over the last second and does not suffer from these problems.

3.3 Example 2: defining and displaying an object

We now extend the first example to define and render an object. In this example, `eg2.c`, we've rearranged the code slightly from Example 1 by introducing some functions. We've done this to keep the code more manageable as we work through the examples.

We'll present the example as a whole and then describe it:

```
/* eg2.c */
#include "maverik.h"

/* Define a box */
void defBox(MAV_box *b)
{
  b->size.x= 1.0; /* Specify its size */
  b->size.y= 2.0;
  b->size.z= 3.0;
  b->matrix= MAV_ID_MATRIX; /* Position and orientation */
  b->sp= mav_sp_default;    /* Surface parameters, i.e. colour */
}

/* Render a frame */
void drawFrame(MAV_SMS *sms)
{
  /* Check for and act on any events */
  mav_eventsCheck();

  /* Request start of a new frame */
  mav_frameBegin();

  /* Display the SMS in all windows */
  mav_SMSDisplay(mav_win_all, sms);

  /* Request end of the frame */
  mav_frameEnd();
}

int main(int argc, char *argv[])
{
  MAV_box box;
  MAV_object *obj;
```

```
  MAV_SMS *sms;

  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  /* Define a box object */
  defBox(&box);

  /* Register the box as a Maverik object */
  obj= mav_objectNew(mav_class_box, &box);

  /* Create a SMS */
  sms= mav_SMSObjListNew();

  /* Add object to SMS */
  mav_SMSObjectAdd(sms, obj);

  /* Rendering loop */
  while (1) drawFrame(sms);
}
```

Example 2 defines a single object – a box. MAVERIK supports 19 different default primitive object classes, including box, sphere, cone, cylinder, polygon and text – Appendix B (page 133) gives the complete list. An application can also define its own new object classes, as described in Chapter 7 (page 67).

This is the MAVERIK data structure to represent a box, `MAV_box` (MFS p 7):

```
  typedef struct {
    MAV_vector size;        /* size of object */
    MAV_surfaceParams *sp;  /* surface parameters */
    MAV_matrix matrix;      /* transformation matrix */
    void *userdef;          /* user-defined data */
  } MAV_box;
```

and comprises of:

- A `MAV_vector` (MFS p 34), `size`, to define the dimensions of the box about its local coordinate system origin. `MAV_vector`'s, comprising of three floats, `x`, `y` and `z`, are used extensively throughout MAVERIK to define 3D vectors.

  MAVERIK, like OpenGL, is intrinsically unitless, in that it does not dictate the use of any particular set of units for its local or world coordinate systems. The choice of units is an arbitrary decision made by the application.

  There are, however, occasions when MAVERIK needs to convert from one set of units into those used by the application. For example, we will see later how the mouse can be used to navigate around the virtual environment. To achieve this MAVERIK needs to convert mouse movements, measured in pixels, into eye position movements, measured in the units chosen by the application. In these cases MAVERIK relies on the application to specify this conversion.

- A `MAV_surfaceParams` (MFS p 29), `sp`, which specifies the "surface parameters", and determines how the object is rendered, enabling the application to specify colour, material characteristics, and texture mapping. In this example we use `mav_sp_default`, the MAVERIK default value for the surface parameters, which renders the box in a pinky-red colour. We'll show how to change the surface parameters in Example 3.

  By default, all objects are drawn filled. You can toggle between filled and wire-frame rendering in a window at any time by pressing **Shift-F8**. MAVERIK responds to a number of function keys at run-time, and the complete set is listed in Appendix A.3 (page 126).

- A `MAV_matrix` (MFS p 81), `matrix`. In MAVERIK, each object is defined in its own private local coordinate system. This is subsequently mapped into the world coordinate system of the virtual environment using the 4x4 transformation matrix specified by this field.

  In the example we have set this to be the identity matrix (`MAV_ID_MATRIX`) so that the box is positioned with its centre at the world coordinate origin and aligned along the major axis.

- The `void *userdef` is a pointer to any extra data an application wishes to attach to the object. We don't use this in this example.

We define the box as follows:

```
void defBox(MAV_box *b)
{
  b->size.x= 1.0;           /* Specify its size */
  b->size.y= 2.0;
  b->size.z= 3.0;
  b->matrix= MAV_ID_MATRIX; /* Position and orientation */
  b->sp= mav_sp_default;    /* Surface parameters, i.e. colour */
}
```

Having defined the box, we now need to register it as a new MAVERIK object:

```
obj= mav_objectNew(mav_class_box, &box);
```

The function `mav_objectNew` (MFS p 179) takes two arguments: the first is an identifier which indicates the **class** of the object – in this case, it's `mav_class_box`, one of the default object classes provided by MAVERIK; the second argument is a pointer to the **data structure** which defines the object.

`mav_objectNew` registers the new object with MAVERIK, and returns a "handle" to the object, which MAVERIK will subsequently use to refer to the object. Note that whatever the class of an object, its handle will always be of the generic object handle type (`MAV_object *`). And because the handle was created by using a pointer to, rather than a copy of, the box object, the handle remains independent of any changes the application makes to the box, such as changing its size.

One of the key aims of MAVERIK is to provide powerful methods for efficiently managing the 3D space of a virtual environment, and the objects which inhabit that space. To achieve this, MAVERIK

introduces the concept of a **Spatial Management Structure** (SMS). An SMS dictates how objects are stored, the culling strategy, level-of-detail processing, and the order in which objects are displayed.

SMS's are, however, too complex an issue to deal with in any depth at this point in this tutorial. At this stage it is sufficient to say that objects must be inserted into an SMS if they are to be displayed.

In the example, we first create a new SMS to manage our virtual environment with `mav_SMSObjListNew` (MFS p 128):

```
MAV_SMS *sms;

sms= mav_SMSObjListNew();
```

which creates a new SMS of type "object list" (we'll explain exactly what this means in a moment). The call returns a "handle" to the SMS, of type `MAV_SMS` (MFS p 111), which is used to refer to it in future calls.

Next we add the box object into the SMS we've just created with `mav_SMSObjectAdd` (MFS p 129):

```
mav_SMSObjectAdd(sms, obj);
```

A similar function, `mav_SMSObjectRmv` (MFS p 130), removes an object from an SMS. Within the main frame loop function `drawFrame` we request display of the SMS in all windows with `mav_SMSDisplay` (MFS p 127):

```
mav_SMSDisplay(mav_win_all, sms);
```

Note that we have not specified any viewing parameters – the eyepoint, the view direction vector, and so on. Unless changed, MAVERIK uses a default set of viewing parameters, with the eyepoint some distance along the positive world-coordinate Z axis looking down that axis towards the origin, with the view-up vector parallel to the world coordinates Y axis. Viewing is described in detail in Chapter 5 (page 47).

The "object list" is the simplest type of SMS and stores objects inserted into it as a simple linked list. When displayed with `mav_SMSDisplay`, this type of SMS uses the axis-aligned bounding box of each object to determine if it is visible.

More complex types of SMS are also provided, such as the "hierarchical bounding box" SMS, which offers a more efficient culling strategy for large models. Users can also make their own SMS's to suit the needs of an application, e.g. one based on cells and portals or one optimized for particular shaped, say long and thin, objects.

Whatever type of SMS is used, the process of creating it always results in the same generic handle: `(MAV_SMS *)`. Therefore, switching between different SMS's is simply a case of changing the single function call which creates the SMS.

When you run Example 2 you should see, after the welcome message has cleared, a blue screen with a red rectangle in the middle. Since the box is viewed edge-on it appears as a rectangle. Quit the program in the same way as for Example 1, by pressing **Shift-Esc**.

3.4 Example 3: Surface parameters and navigation

Our next example, `eg3.c`, demonstrates two more features of MAVERIK: controlling the way an object is rendered using its **surface parameters**, and how to navigate around the virtual environment:

```
/* eg3.c */
#include "maverik.h"
#include <stdio.h>
#include <stdlib.h>

/* Define a box */
void defBox(MAV_box *b, int col)
{
  b->size.x= 1.0; /* Specify its size */
  b->size.y= 2.0;
  b->size.z= 3.0;
  b->matrix= MAV_ID_MATRIX; /* Position and orientation */

  /* Define its "surface parameters", i.e. the colour with which it's rendered */
  /* Use the sign of col to indicate a material or texture, and the value */
  /* of col gives the material or texture index to use */

  if (col>=0)
  {
    b->sp= mav_surfaceParamsNew(MAV_MATERIAL, 0, col, 0); /* Use material index col */
  }
  else
  {
    b->sp= mav_surfaceParamsNew(MAV_TEXTURE, 0, 0, -col); /* Use texture index col */
  }
}

/* Render a frame */
void drawFrame(MAV_SMS *sms)
{
  /* Check for and act on any events */
  mav_eventsCheck();

  /* Request start of a new frame */
  mav_frameBegin();

  /* Display the SMS in all windows */
  mav_SMSDisplay(mav_win_all, sms);

  /* Request end of the frame */
```

```
  mav_frameEnd();
}

int main(int argc, char *argv[])
{
  MAV_box box;
  MAV_object *obj;
  MAV_SMS *sms;

  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  if (argc != 2) {
    printf("usage: %s colour\n", argv[0]);
    exit(1);
  }

  /* Define a box object */
  defBox(&box, atoi(argv[1]));

  /* Use default mouse navigation */
  mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);

  /* Register the box as a Maverik object */
  obj= mav_objectNew(mav_class_box, &box);

  /* Create a SMS */
  sms= mav_SMSObjListNew();

  /* Add object to SMS */
  mav_SMSObjectAdd(sms, obj);

  /* Rendering loop */
  while (1) drawFrame(sms);
}
```

In `eg3.c`, we've extended the `defBox` function to take an argument, `col`, which is used to control the object's surface parameters, i.e. the colour with which it is rendered.

Every MAVERIK window has a "palette" associated with it. This contains a colour table, material table, texture table, font table, and light table, each of which is intialised with a number of default values when the window is created. An object's "surface parameters" specify which table entries in the palette to use when rendering the object.

In Example 2, we used `mav_sp_default` as the surface parameters; here, we define the surface parameters using `mav_surfaceParamsNew` (MFS p 207):

```
  MAV_surfaceParams *mav_surfaceParamsNew (int mode, int colour,
                                           int material, int texture);
```

This function creates a new set of surface parameters. Depending on the value of `mode`, objects may be rendered with a simple colour, a material type, a texture, or a combination of these – see Section 6.1 (page 55) for details. The remaining values, `colour`, `material` and `texture`, specify which entry or entries in the palette to use. Rarely does more than one of these three values need to be given, and values which are not applicable should be set to zero.

In the example, if `col` is positive, it's used to select a material from the window's material table; if it's negative, it selects a texture:

```
if (col>=0)
{
   b->sp= mav_surfaceParamsNew(MAV_MATERIAL, 0, col, 0); /* Use material index col */
}
else
{
   b->sp= mav_surfaceParamsNew(MAV_TEXTURE, 0, 0, -col); /* Use texture index col */
}
```

The other MAVERIK feature introduced in this example is "navigation". Navigation is an example of the "application behaviour" aspect of a MAVERIK program, and is enabled by calling the function `mav_navigationMouse` (MFS p 174):

```
mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);
```

This activates the default navigation method in all active windows, controlled by the desktop mouse, as follows:

- With the left mouse button pressed, mouse movement translates the eyepoint forwards/backwards, and yaws (rotates about the Y axis) the view.
- With the right mouse button pressed, mouse movement translates the eyepoint up/down and left/right;

In case you're wondering how this works, navigation is actually triggered by the `mav_eventsCheck` function. In Chapter 5 (page 47) we discuss navigation in detail, listing the various navigation methods avaliable to the application. You can also create your own customised kinds of navigation which is described in Chapter 8 (page 91).

To execute this example you have to provide an integer on the command line to determine which material or texture to use, e.g "eg3 5" uses default material 5 (a white-ish colour), "eg3 -1" uses default texture 1 (a marble effect). There are 20 default materials (numbered 0–19) with number 1 being used to make the pinky-red default set of surface parameters. There are 2 default textures (numbers 1 and 2). Section 6.1 (page 55) describes how to specify your own colours, materials and textures.

The initial view should be the same as the last example (except it will be a different coloured rectangle), but now you will be able to move around the box using the mouse commands described above. Quit the example in the usual manner.

3.5 Example 4: a more complex environment

Our next example, `ex4.c`, draws together features we have seen in the previous examples, to create a more complex virtual environment, comprising a number of different classes of object in random positions and orientations, with random surface parameters, and a textured ground plane.

This example demonstrates:

- defining other classes of object: a rectangle, cylinder and composite object;
- using the `matrix` field of an object to set its position and orientation;
- defining a texture map from a file;
- populating the virtual environment with multiple objects.

```
/* eg4.c */
#include "maverik.h"
#include <stdio.h>
#include <stdlib.h>

MAV_surfaceParams *sp[4];

/* Define a rectangle */
void defRect(MAV_rectangle *r)
{
  r->width= 500.0; /* Size */
  r->height= 500.0;
  r->xtile= 3; /* Texture repeat tiling */
  r->ytile= 3;
  /* Orientation (RPY 0,-90,0) and position (XYZ 0,-2,0) */
  r->matrix= mav_matrixSet(0,-90,0, 0,-2,0);

  /* Use decal texture with index 5 */
  r->sp= mav_surfaceParamsNew(MAV_TEXTURE, 0, 0, 5);
}

/* Define a box */
void defBox(MAV_box *b)
{
  /* Random box size, position/orientation and set of surface params */
  b->size.x= mav_random()*30;
  b->size.y= mav_random()*30;
  b->size.z= mav_random()*30;
```

```
  b->matrix= mav_matrixSet(0,0,mav_random()*360,
                           -200+mav_random()*400,0,-200+mav_random()*400);
  b->sp= sp[(int) (mav_random()*4)];
}

/* Define a cylinder */
void defCyl(MAV_cylinder *c)
{
  /* Random cylinder size, position/orientation and set of surface params */
  c->radius= mav_random()*20;
  c->height= mav_random()*20;
  c->endcap= 1;
  c->nverts= 10;
  c->matrix= mav_matrixSet(0,mav_random()*360,0,
                           -200+mav_random()*400,0,-200+mav_random()*400);
  c->sp= sp[(int) (mav_random()*4)];
}

/* Define a composite object */
void defComp(MAV_composite *c)
{
  /* Read AC3D object from file */
  if (!mav_compositeReadAC3D("mavlogo.ac", c, MAV_ID_MATRIX)) {
    printf("failed to read mavlogo.ac\n");
    exit(1);
  }

  /* Fixed position and orientation */
  c->matrix= mav_matrixSet(0,0,0, 0,0.2,-15);
}

/* Render a frame */
void drawFrame(MAV_SMS *sms)
{
  /* Check for and act on any events */
  mav_eventsCheck();

  /* Request start of a new frame */
  mav_frameBegin();

  /* Display the SMS in all windows */
  mav_SMSDisplay(mav_win_all, sms);

  /* Request end of the frame */
  mav_frameEnd();
}

int main(int argc, char *argv[])
{
  MAV_rectangle gp;
  MAV_SMS *objs;
  MAV_box box[10];
  MAV_cylinder cyl[10];
```

```
  MAV_composite comp;
  int i;

  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  /* Define a texture map from file, texture index 5 */
  mav_paletteTextureSet(mav_palette_default, 5, "marble_floor.ppm");

  /* Define a set of "surface parameters", i.e. the colour with */
  /* which an object is rendered */
  sp[0]= mav_surfaceParamsNew(MAV_MATERIAL, 0, 1, 0); /* Material index 1 */
  sp[1]= mav_surfaceParamsNew(MAV_MATERIAL, 0, 2, 0); /* Material index 2 */
  sp[2]= mav_surfaceParamsNew(MAV_MATERIAL, 0, 3, 0); /* Material index 3 */
  /* Texture 1 modulated with material 2 */
  sp[3]= mav_surfaceParamsNew(MAV_LIT_TEXTURE,  0, 2, 1);

  /* Define a rectangle to act as the ground plane */
  defRect(&gp);

  /* Create an SMS for the objects and add the ground plane to it */
  objs= mav_SMSObjListNew();
  mav_SMSObjectAdd(objs, mav_objectNew(mav_class_rectangle, &gp));

  /* Create 10 boxes and cylinders */
  for (i=0; i<10; i++) {

    /* Define a box and a cylinder */
    defBox(&box[i]);
    defCyl(&cyl[i]);

    /* Add the box and cylinder to the objs SMS */
    mav_SMSObjectAdd(objs, mav_objectNew(mav_class_box, &box[i]));
    mav_SMSObjectAdd(objs, mav_objectNew(mav_class_cylinder, &cyl[i]));
  }

  /* Define a composite object and add it to objs SMS */
  defComp(&comp);
  mav_SMSObjectAdd(objs, mav_objectNew(mav_class_composite, &comp));

  /* Use default mouse navigation */
  mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);

  /* Rendering loop */
  while (1) drawFrame(objs);
}
```

The example begins by defining from file `marble_floor.ppm` entry number 5 in the default palette's texture table. All windows are associated with the default palette (`mav_palette_default`) unless explicitly re-assigned. MAVERIK can read textures defined in virtually any image file format since it uses ImageMagick's `convert` program to convert the file into PPM format which is trivial to parse. N.B. if ImageMagick is not installed then you will be limited to PPM image files.

It then initialises an array of surface parameters. Array indices 0–2 are set to be materials from the default set, while index 3 is default texture 1 modulated with default material 2 (effectively giving a lit texture).

Next, new objects are defined by the functions `defRect`, `defBox`, `defCyl` and `defComp`. `defRect` defines a textured rectangle:

```
r->width= 500.0; /* Size */
r->height= 500.0;
r->xtile= 3; /* Texture repeat tiling */
r->ytile= 3;
/* Orientation (RPY 0,-90,0) and position (XYZ 0,-2,0) */
r->matrix= mav_matrixSet(0,-90,0, 0,-2,0);
/* Use decal texture with index 5 */
r->sp= mav_surfaceParamsNew(MAV_TEXTURE, 0, 0, 5);
```

The rectangle object, fully described in Section B.14 (page 148), is centered at its local coordinate frame origin and defined by a `width` along its local coordinate frame X axis and a `height` along its Y axis.

The rectangle is defined in the local coordinate frame XY plane with its normal along the positive Z axis, but we want to use it in this example to represent the ground plane which is the world coordinate frame XZ plane with a normal along the Y axis. So, the transformation between local and world coordinate frames needs to rotate the rectangle by 270 (or -90) degrees about its local coordinate frame X axis. Such a transformation would place the rectangle on the world coordinate frame XZ plane at $Y = 0$. (MAVERIK uses a right handed coordinate system and so a rotation of just 90 degrees would place the rectangle in the XZ plane, but with its normal aligned with the negative Y axis. We need to rotate it a further 180 degrees in order for its normal to be correctly oriented.)

The default eyepoint is also at $Y = 0$ and therefore the rectangle would not be visible since we would be viewing it exactly along the plane. To overcome this, the rectangle needs to be offset by some amount along the negative Y axis so it appears beneath us. The same effect could more correctly be achieved by moving the eyepoint upwards, and we show how to perform this is Chapter 5 (page 47).

This transformation matrix to achive this is defined using the function `mav_matrixSet` (MFS p 156):

```
MAV_matrix mav_matrixSet(float roll, float pitch, float yaw,
                         float x, float y, float z);
```

where `roll`, `pitch` and `yaw`, are defined to be rotation, in degrees, about the Z, X and Y axes respectively. Rotations are applied in the order roll, yaw, then pitch.

So the rectangle's matrix is set as follows:

```
r->matrix= mav_matrixSet(0,-90,0, 0,-2,0);
```

This places it the world coordinate frame XZ plane at $Y = -2$.

The box and cylinder objects are given a random size, position, orientation and surface parameters, as `defCyl` illustrates:

```
c->radius= mav_random()*20;
c->height= mav_random()*20;
c->endcap= 1;
c->nverts= 10;
c->matrix= mav_matrixSet(0, mav_random()*360, 0,
                         -200+mav_random()*400, 0, -200+mav_random()*400);
c->sp= sp[(int) (mav_random()*4)];
```

where `mav_random` (MFS p 203) returns a pseudo-random number in the range zero to one.

3.5.1 Level of detail

The cylinder in this example is rendered with `nverts` facets. If you set the option variable `mav_opt_curveLOD` to `MAV_TRUE`, then MAVERIK will ignore the `nverts` value and render the cylinder with as many, or as few, facets as it deems necessary to accurately represent the curved surface. However, it will never use more than `mav_opt_vertsMax` or less than `mav_opt_vertsMin` facets. Furthermore, the rate at which the number of vertices used is reduced as the object recedes from the eye point, is controlled by the arbitrary constant `mav_opt_curveFactor`. An undocumented example in the `examples/misc/LOD` sub-directory of the MAVERIK distribution allows you to dynamically change these variables and observe the effects.

Note however that even if you are using automatic level of detail, `nverts` must be set to a valid value (i.e. greater than 2) since it is used for other purpose besides rendering, such as in calculating the cylinder's bounding box.

3.5.2 Reading objects from file

As well as its simple primitive object classes, MAVERIK also supports a "composite object", which comprises a set of other objects linked together. Although an application can define composite objects "by hand", MAVERIK provides a convenient way to create them automatically, by reading object definitions from AC3D [1], VRML97 or Lightwave format files.

AC3D is an interactive geometry modeler which, as well as creating and editing objects, can import objects defined in a number of common 3D file formats (including 3DS, DXF, Lightwave and VRML1). `defComp` defines a "composite object", read in from the AC3D file `mavlogo.ac` (it's a 3D MAVERIK logo). See Section B.18 (page 152) for full details of composite objects.

We'll return to this example in Chapter 4.

3.6 Summary

We hope these four simple examples have given you an insight into how to create simple applications with MAVERIK. In subsequent chapters we'll cover MAVERIK's functionality in more detail, and present more example programs to illustrate more advanced techniques.

Chapter 4

Keyboard and mouse events

In MAVERIK, an application defines the actions to be taken when mouse and keyboard events occur using a generalised callback mechanism. In this chapter we describe how MAVERIK handles input events, how these relate to objects, and how to write and register callback functions.

4.1 Example 5: basic event handling

This example, `eg5.c`, expands Example 2 from Section 3.3 (page 22) so that when the middle mouse button is pressed while the cursor is pointing at the box, it increases in size, and when a key is pressed a message is printed to the shell window. Note that the navigation has also been included in this example.

If you don't have a middle mouse button, and can't emulate one on your operating system, then it is trivial to modify this example to work with either the left or right buttons. However, note that the left and right buttons will also trigger navigation.

```
/* eg5.c */
#include "maverik.h"
#include <stdio.h>

/* Define a box */
void defBox(MAV_box *b)
{
  b->size.x= 1.0; /* Specify its size */
  b->size.y= 2.0;
  b->size.z= 3.0;
  b->matrix= MAV_ID_MATRIX; /* Position and orientation */
  b->sp= mav_sp_default;    /* Surface parameters, i.e. colour */
}

/* Render a frame */
void drawFrame(MAV_SMS *sms)
```

```
{
  /* Check for and act on any events */
  mav_eventsCheck();

  /* Request start of a new frame */
  mav_frameBegin();

  /* Display the SMS in all windows */
  mav_SMSDisplay(mav_win_all, sms);

  /* Request end of the frame */
  mav_frameEnd();
}

/* Mouse event callback */
int mouseEvent(MAV_object *o, MAV_mouseEvent *me)
{
  MAV_box *box;

  /* Convert from generic Maverik object to the box object */
  box= (MAV_box *) mav_objectDataGet(o);

  if (me->movement==MAV_PRESSED) { /* Only consider button presses */
    box->size.x+=1.0; /* Make box a bit bigger */
  }

  return 1;
}

/* Keyboard event callback */
int keyEvent(MAV_object *o, MAV_keyboardEvent *ke)
{
  if (ke->movement==MAV_PRESSED) { /* Only consider button presses */
    if (ke->key<255) { /* Only consider printable ASCII characters */
      printf("Pressed %c\n",  ke->key);
    }
  }

  return 1;
}

int main(int argc, char *argv[])
{
  MAV_box box;
  MAV_object *obj;
  MAV_SMS *sms;

  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  /* Define a box object */
  defBox(&box);
```

```
  /* Register the box as a Maverik object */
  obj= mav_objectNew(mav_class_box, &box);

  /* Create a SMS */
  sms= mav_SMSObjListNew();

  /* Add object to SMS */
  mav_SMSObjectAdd(sms, obj);

  /* Use default mouse navigation */
  mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);

  /* Define mouse callback */
  mav_callbackMouseSet(MAV_MIDDLE_BUTTON, mav_win_all, mav_class_box, mouseEvent);

  /* Define keyboard event callback */
  mav_callbackKeyboardSet(mav_win_all, mav_class_world, keyEvent);

  /* Rendering loop */
  while (1) drawFrame(sms);
}
```

Mouse and keyboard event callbacks are defined with the functions `mav_callbackMouseSet` (MFS p 133) and `mav_callbackKeyboardSet` (MFS p 132) respectively. Mouse event callbacks are defined for a specific button, while keyboard event callbacks are defined for any key.

In addition, event callbacks are defined on a per-window and **per-object-class**, rather than **per-object**, basis. This means that, for example, all boxes will share the same event callback function. This may seem unusual at first, but it is a fundamental way in which MAVERIK deals with objects. It would be trivial to implement a per-object event callback mechanism by having the per-object-class callback function execute another function which was stored in the object's data structure and setting this to be a different function for different objects.

Setting the callback on a per-window basis allows for objects to respond differently to events in different windows (we shall see in Section 6.2.6 (page 64) how to open multiple windows). However, here we use the "all windows" identifier `mav_win_all` to set the event callback.

In this example a mouse event callback function (`mouseEvent`) will be called for middle mouse button events which occur while the mouse is pointing at any box.

The keyboard event callback function (`keyEvent`) is set for the primitive class `mav_class_world`. Callbacks set for this class are activated when an event occurs anywhere in the window. Two similar classes also exist: `mav_class_any` and `mav_class_miss`. Respectively, these allow callbacks to be defined for events which occur when the mouse is over any object, regardless of its class, and when the mouse is over no object.

The callback functions take as their arguments the MAVERIK object which the mouse was over when the event occurred, and a data structure which details the event. No attempt should be made to interpret the MAVERIK object passed to the callback function set for the classes `mav_class_world`, `mav_class_any` and `mav_class_miss`.

At this stage we will ignore the significance of the return value of the callback function, and return to this in Section 4.3 (page 40).

The first action the mouse event callback function needs to perform is to convert the MAVERIK object which it receives in its `o` argument, into the box object so it can operate on it. The function which does this, `mav_objectDataGet` (MFS p 176), simply returns the object data pointer maintained by the MAVERIK object. However, this is returned as a generic `void` pointer, rather than a pointer to a box object, but since this callback will only be activated for MAVERIK objects which are of the box class (observe how the callback was set), we can safely cast this pointer into a box object pointer.

The keyboard event callback function simply prints a message in the shell window indicating which key was pressed. This data is the `key` field of the `MAV_keyboardEvent` (MFS p 17) data structure which details the event. Note that non-ASCII symbols, such as the pound and euro signs, may not correctly interpreted.

4.2 Example 6: modifying the rendering loop

Now we have introduced some of the subtleties of event callbacks with a simple example, we return to the "ground plane and objects" of Example 4 (page 29). We now allow the user to increase the radius of a cylinder and scale of the composite object (the MAVERIK logo) by clicking on these objects with the middle mouse button. The keyboard event function traps two key presses: 'q' which quits the application and 'h' which displays a help message.

The following is excerpted from `eg6.c`, and shows how the event functions are used:

```
/* eg6.c [excerpt] */

/ * code omitted */

/* Mouse event for cylinders */
int cylEvent(MAV_object *obj, MAV_mouseEvent *ev)
{
  MAV_cylinder *cyl;

  /* Convert from generic Maverik object to a cylinder object */
  cyl= (MAV_cylinder *) mav_objectDataGet(obj);

  if (ev->movement==MAV_PRESSED) { /* Only consider button presses */
    cyl->radius+=1; /* Increase cylinder radius */
  }

  return 1;
}

/* Mouse event for composites */
int compEvent(MAV_object *obj, MAV_mouseEvent *ev)
{
```

```
  if (ev->movement==MAV_PRESSED) {
    MAV_composite *comp= (MAV_composite *) mav_objectDataGet(obj);
    /* Scale composite by a factor of 1.1 */
    comp->matrix= mav_matrixScaleSet(comp->matrix, 1.1);
  }

  return 1;
}

/* Display a help message */
void helpMsg(void *ignored)
{
  mav_stringDisplay(mav_win_all, "Left mouse button - navigate forward/backward and yaw",
                    MAV_COLOUR_BLACK, 0, -0.95, 0.90);
  mav_stringDisplay(mav_win_all, "Right mouse button - navigate up/down and left/right",
                    MAV_COLOUR_BLACK, 0, -0.95, 0.83);
  mav_stringDisplay(mav_win_all, "Middle mouse click on cylinder - increase radius",
                    MAV_COLOUR_BLACK, 0, -0.95, 0.76);
  mav_stringDisplay(mav_win_all,
                    "Middle mouse click on composite (Maverik logo) - increase scale",
                    MAV_COLOUR_BLACK, 0, -0.95, 0.69);
  mav_stringDisplay(mav_win_all, "h - help", MAV_COLOUR_BLACK, 0, -0.95, 0.60);
  mav_stringDisplay(mav_win_all, "q - quit", MAV_COLOUR_BLACK, 0, -0.95, 0.53);
}

/* Keyboard event */
int keyEvent(MAV_object *obj, MAV_keyboardEvent *ke)
{
  switch (ke->key) {
  case 'q': /* Quit */
    exit(1);
    break;

  case 'h': /* Help */
    if (ke->movement==MAV_PRESSED)
    {
      /* Begin executing function helpMsg at the end of each frame */
      mav_frameFn3Add(helpMsg, NULL);
    }
    else
    {
      /* Stop executing function helpMsg at the end of each frame */
      mav_frameFn3Rmv(helpMsg, NULL);
    }
    break;
  }

  return 1;
}

int main(int argc, char *argv[])
{
  /* code omitted */
```

```
/* Define mouse event callbacks */
mav_callbackMouseSet(MAV_MIDDLE_BUTTON, mav_win_all, mav_class_cylinder, cylEvent);
mav_callbackMouseSet(MAV_MIDDLE_BUTTON, mav_win_all, mav_class_composite, compEvent);

/* Define keyboard event callback */
mav_callbackKeyboardSet(mav_win_all, mav_class_world, keyEvent);

/* code omitted */
```

This example shows how we can modify the rendering loop by dynamically adding and removing application-defined functions which are executed at various stages in the rendering loop.

Rendering a frame can be broken down into 3 phases:

- **Phase 1:** before the window is cleared and the view for the frame is fixed;
- **Phase 2:** the window is now cleared, the view is fixed, but no objects have yet been drawn;
- **Phase 3:** all objects have now been drawn and the frame is complete, but the buffers have not yet been swapped.

The functions `mav_frameFn1Add` (MFS p 142) and `mav_frameFn1Rmv` (MFS p 142) respectively add and remove functions to be executed at phase 1; there are corresponding functions for the other rendering phases, named `mav_frameFnNAdd` (where `N` is 1, 2 or 3). There is no limit on the number of functions which can be added to each phase. The second argument to `mav_frameFnNAdd` is not interpreted by MAVERIK, rather it forms the single parameter to the application defined function thus allowing data to be passed into the function. This feature is not used in this example.

Example 6 adds the function `helpMsg` to be executed at phase 3 when the 'h' key is pressed. When that key is released the function is removed. `helpMsg` prints a help message on screen using the function `mav_stringDisplay` (MFS p 205) which takes as its arguments the window on which it acts, the string to display, the colour and font to use and where to position the text. This text is not a 3D object in the world, but rather "annotation text" overlayed on the 2D window. The position of the string is given as an x,y position where $(-1,-1)$ maps to the bottom left of the screen and $(1,1)$ to the top right.

Note that either of phases 2 or 3 would suffice to display this message, but if phase 1 was used nothing would have been displayed since the message would have been rendered before the window was cleared.

4.3 Example 7: advanced event handling

We now extend Example 6 to demonstrate more advanced event handling and the use of so-called "process-based" callbacks. This term refers to callbacks which perform arbitrary operations on objects – it does not refer to "processes" in the Unix sense of the word.

The following code is excerpted from `eg7.c`.

```
/* eg7.c [excerpt] */

/* code omitted */

MAV_matrix *objMat1, *objMat2;
float objDist;
int fc=0;

/* Function to make object jump */
void jump(void *ignored)
{
  /* Increase Y component of matrix by an ammount which ranges
     +4 to -4 over 60 interactions */
  objMat1->mat[MAV_MATRIX_YCOMP]+=cos(MAV_DEG2RAD(fc*3.0))*4.0;

  /* Stop executing this function after 60 frames */
  fc++;
  if (fc>60) {
    fc=0;
    mav_frameFn1Rmv(jump, NULL);
  }
}

/* Function to drag object with mouse */
void pick(void *ignored)
{
  MAV_vector pos;

  /* Calculate the position of a point a distance objDist away from the eye along */
  /* the normalized vector defined by the eye point and the mouse's projection */
  /* onto the near clip plane (this is mav_mouse_dir) */
  pos= mav_vectorAdd(mav_win_current->vp->eye, mav_vectorScalar(mav_mouse_dir, objDist));

  /* Set the object's matrix to this position */
  *objMat2= mav_matrixXYZSet(*objMat2, pos);
}

/* Keyboard event */
int keyEvent(MAV_object *obj, MAV_keyboardEvent *ke)
{
  MAV_surfaceParams **spptr;

  switch (ke->key) {
  case 'q': /* Quit */
    exit(1);
    break;

  case 'h': /* Help */
    if (ke->movement==MAV_PRESSED)
    {
```

```
    /* Begin executing function helpMsg at the end of each frame */
    mav_frameFn3Add(helpMsg, NULL);
  }
  else
  {
    /* Stop executing function helpMsg at the end of each frame */
    mav_frameFn3Rmv(helpMsg, NULL);
  }
  break;
}

/* Only consider event if the mouse was pointing at an object */
if (ke->intersects) {
  if (ke->movement==MAV_PRESSED) { /* Only consider button press event */
    switch (ke->key) {
    case 'd':  /* Delete an object */
      mav_objectDelete(ke->obj);
      break;

    case 'b': /* Increase size of box */
      /* Ensure object is a box */
      if (mav_objectClassGet(ke->obj)==mav_class_box)
      {
        /* Convert from generic Maverik object to a box object */
        MAV_box *box= (MAV_box *) mav_objectDataGet(ke->obj);
        box->size.x+=0.5; /* Increase size of box */
      }
      else
      {
        printf("Object is not a box\n");
      }
      break;

    case 'c': /* Change colour */
      if (mav_callbackGetSurfaceParamsExec(mav_win_current, ke->obj, &spptr)) {
        /* Get a ptr to the surfaceParmas field of the object */
        *spptr= sp[(int) (mav_random()*4)]; /* Set it to some random value */
      }
      break;

    case 'j': /* Make object jump */
      if (fc==0) { /* Only if something is not currently in flight */
        /* Get a ptr to the matrix field of the object */
        if (mav_callbackGetMatrixExec(mav_win_current, ke->obj, &objMat1)) {
          /* Begin executing function jump at the start of each frame */
          mav_frameFn1Add(jump, NULL);
        }
      }
      break;
    }
  }

  switch (ke->key) {
```

```
    case 'p': /* Pick object */
      if (ke->movement==MAV_PRESSED)
      {
        /* Get a ptr to the matrix field of the object */
        if (mav_callbackGetMatrixExec(mav_win_current, ke->obj, &objMat2)) {
          /* Remember distance from eye to object intersection */
          objDist= ke->objint.pt1;
          /* Begin executing function pick after the view has been set */
          mav_frameFn2Add(pick, NULL);
        }
      }
      else
      {
        /* Stop executing function pick after the view has been set */
        mav_frameFn2Rmv(pick, NULL);
      }
      break;
    }
  }

  return 1;
} /* keyEvent */

int main(int argc, char *argv[])
{

  /* code omitted */

  /* Create an SMS for the ground plane and add rectangle object to it */
  groundPlane= mav_SMSObjListNew();
  mav_SMSObjectAdd(groundPlane, mav_objectNew(mav_class_rectangle, &gp));

  /* Make objects in groundPlane SMS unselectable to keyboard and mouse event */
  mav_SMSSelectabilitySet(groundPlane, mav_win_all, MAV_FALSE);

  /* code omitted */
}
```

Recall that the keyboard event callback is registered for the `mav_class_world` class and as such the MAVERIK object passed to the callback function should not be interpreted. However, stored in both the keyboard and mouse event data structures is the object which the mouse was over when the event occurred. This information is stored in the `obj` field of the data structure along with `intersects` which indicates if the cursor was pointing at an object.

In Example 7, pressing the ‘d’ key deletes the object the mouse is pointing at. This is achieved by calling `mav_objectDelete` (MFS p 178) which removes the object from any SMS’s which it is in before deleting the object.

By default, any object which is in an SMS can activate the event callbacks. However, an SMS can be set to be “non-selectable” and objects in such an SMS will not trigger event callbacks if the

mouse was pointing at them. So, by having multiple SMS's an application can maintain groups of objects which are selectable, and groups which are not. Example 7 uses two SMS's: one for the ground plane, and one for the remaining objects. The SMS containing the ground plane object is set to be non-selectable and so pointing at this object will not trigger keyboard and mouse events. Note: setting the selectability of an SMS also determines whether it is searched in the functions which check whether a line or bounding box intersect any objects – `mav_SMSIntersectLineAll` (MFS p 239) and `mav_SMSIntersectBBAll` (MFS p 237) respectively.

Pressing 'b' increases the size of a box. This is similar to Example 5 (page 35) except that now we first have to ensure that the object pointed to by the mouse really is a box. This is achieved with the function `mav_objectClassGet` (MFS p 175) which returns the class of an object.

4.3.1 Process-based callbacks

The 'c' key changes the set of surface parameters which are used to render the object which the mouse is pointing at. All of the default MAVERIK primitives have this field, named `sp`, in their data structure. The problem is: how can we access this field when we only have a generic MAVERIK object to work with?

One way would be to have a large switch statement which checked the class of the MAVERIK object (using the function described above for the 'b' key) and then casts the data portion of the MAVERIK object to be the data structure appropriate for this type of primitive, thus allowing direct access to the required field.

Alternatively, it could be arranged that for each class of object there was a function which returned a pointer to its surface parameters field (a pointer being more useful since it can be used to change the value stored in the data structure). The function would take as input a MAVERIK object, cast the data portion of this to be the relevant data structure for the class of object and return a pointer to the required field. Furthermore, if this function was accessible via the MAVERIK class data structure, then there is enough information encapsulated in a MAVERIK object to execute the relevant function and gain access to the surface parameters field.

While appearing overly complicated at first, the second method is, in fact, preferable since it allows for new classes of objects to be seamlessly added. (Using the first method you would have to extend the switch statement to accommodate the new class). The ability to add new classes of objects is a key aspect of MAVERIK and we show how this is performed in Chapter 7.

Essentially this is a callback mechanism and it can be thought of as being analogous to the event-based callbacks introduced earlier in this chapter. To distinguish between the two, we call the type of callback just introduced "process-based" since they perform an arbitrary processing operation on a object – such as accessing a specific data field. A conceptual difference between the two is that event-based callbacks are executed by MAVERIK, whereas process-based callbacks are explicitly invoked by the application itself. However, this does not prevent them being implemented with the same mechanism.

The function `mav_callbackGetSurfaceParamsExec` (MFS p 259) executes the "get surface param-

eters" process-callback on an object. Its full prototype is:

```
int mav_callbackGetSurfaceParamsExec(MAV_window *w, MAV_object *o,
                                     MAV_surfaceParams ***sp);
```

Let's break this function prototype down into each parameter.

The first argument, `w`, is a MAVERIK window and indicates which window the callback is being executed for. This seems very strange at first, but recall from earlier in this chapter that event-based callbacks are defined on a per-window and per-object class basis (thus allowing objects the ability to respond to mouse event differently in different windows). And, as the two different types of callback are implemented by the same mechanism, process-based callbacks are also defined on the same basis. That said, the authors cannot envisage the case where, for example, the "get surface parameters" callback function would be implemented differently in different windows!

The second argument, `o`, is the MAVERIK object in question.

The third argument is a triple pointer to a MAVERIK surface parameters data structure – actually, it's the address of a pointer to the required field in the data structure. Put another way, recall that a pointer to the desired field in the data structure is required so that its value can be set. Unfortunately, the required field is itself a pointer, and so a pointer to this pointer is needed. Furthermore, this value can not be passed back as the functions return value since that is used for another purpose (see below). Therefore, the only option is to pass into the function the address of a pointer to the required field so that the function can set the contents of this address to be the appropriate value. Hopefully, its use in Example 7 (summarized below) will help clarify this:

```
MAV_surfaceParams **spptr;
if (mav_callbackGetSurfaceParamsExec(mav_win_current, ke->obj, &spptr)) {
  *spptr= sp[(int) (mav_random()*4)]; /* Set it to some random value */
}
```

Note the use of the "current window" handle `mav_win_current` to specify the window the callback is being executed for. Virtually all process-based callbacks are executed in this manner.

The return value of this function call is `MAV_TRUE` or `MAV_FALSE` and indicates if the callback was successful. There are two reasons why the execution of a callback can fail: either there is no callback function provided for this class of object, or the callback function could not successfully complete the operation for some reason.

The 'j' key makes the object pointed at "jump in the air". This is achieved by using the "get matrix" process-based callback to obtain a pointer to the transformation matrix of the object under the cursor. The `Y` position component of this matrix is then manipulated by a `mav_frameFn1` function to move the object vertically up 4 units and then back down by the same amount over 60 consecutive frames. One point worth noting in the implementation of this is that the frame function automatically removes itself after the 60 frames have elapsed.

Holding down the 'p' key allows the user to drag an object around the scene with the mouse.

In order to describe how this is achieved we have to introduce the concept of the mouse's 3D world position. While the desktop mouse is intrinsically a 2D device, its position can be mapped onto the near clip plane to give it a 3D position in the world. A vector can be defined using the eyepoint and this position. It is the intersection of any objects with this vector thats allows MAVERIK to determine if the mouse is pointing at any object. This vector turns out to be very useful and so is calculated by MAVERIK at the start of each frame and stored in the global variable `mav_mouse_dir`.

Back to moving objects around. When the key press event occurs, the distance from the eye to the first point of intersection on the objects surface is noted. This value, `pt1`, is part of the `MAV_objectIntersection` (MFS p 84) data structure, `objint`, which itself is part of both the keyboard and mouse event data structures. In addition, the "get matrix" process-based callback is executed on the selected object and a `mav_frameFn1` function is added.

This function sets the positional part of the transformation matrix so that the object is maintained at the noted distance from the eyepoint along the vector described above.

When the 'p' key is released the frame function is removed and the object stops following the cursor.

Chapter 5

Viewing and navigation

So far the examples have used the default viewpoint and mouse navigation. In this chapter we will show how an application can override these defaults and specify its own.

5.1 View parameters

The MAVERIK viewing model is based on the standard computer graphics viewing model, where the application defines:

- an **eyepoint** – the position of the eye in world coordinates;
- a **view direction** – a normalized vector indicating the direction of view from the eyepoint;
- the **view up** – a normalized vector defining the viewer's "up" direction.

In addition, an application also needs to define the following:

- the **World up (or fixed up) vector** – a normalized vector indicating the direction of the World "up" direction. This is not actually used to define the view but needed by the default navigation.
- the **view modifier function** – the meaning of which can be ignored for now (it is described in Chapter 11.2 (page 114)) but we need to mention it because its value must explicitly be set to `NULL`.

Collectively, all the above data is called the "view parameters" and are stored in the `MAV_viewParams` (MFS p 36) data structure:

```
typedef struct {
```

```
    MAV_vector eye;         /* eyepoint */
    MAV_vector view;        /* view direction vector */
    MAV_vector up;          /* view up vector */
    MAV_vector fixed_up;    /* world up vector  */
    MAV_viewModifierFn mod; /* view modifier function */
} MAV_viewParams;
```

The data structure actually contains additional fields, but we'll ignore them for now.

We'll describe how to define the parameters for a perspective view (the horizontal and vertical fields of view), in Chapter 6.

Each MAVERIK window has an associated set of view parameters which, by default, is `mav_vp_default` which is:

```
eye=      (0, 0, 10);   /* eye on the positive z-axis... */
view      (0, 0, -1);   /* ...looking down z-axis towards origin */
up=       (0, 1, 0);    /* "up" direction parallel to world y-axis */
fixed_up= (0, 1, 0);    /* "world up" is world y-axis */
mod=      NULL;         /* no view modifier function */
```

A window can be associated with a different set of view parameters with the function `mav_windowViewParamsSet` (MFS p 232):

```
void mav_windowViewParamsSet(MAV_window *w, MAV_viewParams *vp);
```

where `w` is the window in question and `vp` a pointer to the view parameters to use. (Since a pointer is used the window does not need to be notified if the values of the view parameters subsequently change).

A pointer to the view parameters associated with a window is held in the `vp` field of the `MAV_window` (MFS p 94) data structure.

Example 8 (`eg8.c`) modifies Example 7 (page 40) to define a set of view parameters, and to associate them with all windows, as follows:

```
MAV_viewParams vp;

/* Define initial view parameters */
vp.eye.x= 0;    /* Eyepoint */
vp.eye.y= 25;
vp.eye.z= 200;

vp.view.x= 0;   /* View direction */
vp.view.y= 0;
vp.view.z= -1;
```

```
vp.up.x= 0;     /* View up direction */
vp.up.y= 1;
vp.up.z= 0;

vp.fixed_up= vp.up;  /* World up direction */
vp.mod= NULL;        /* No view modification required */

/* Use these view parameters in all windows  */
mav_windowViewParamsSet(mav_win_all, &vp);
```

The keyboard event callback has been modified in this example to trap the ‘r’ key which will reset the view parameters to their initial values as follows:

```
mav_win_current->vp->eye.x= 0;    /* eyepoint */
mav_win_current->vp->eye.y= 25;
mav_win_current->vp->eye.z= 200;

mav_win_current->vp->view.x= 0;   /* View direction */
mav_win_current->vp->view.y= 0;
mav_win_current->vp->view.z= -1;

mav_win_current->vp->up.x= 0;     /* View up direction */
mav_win_current->vp->up.y= 1;
mav_win_current->vp->up.z= 0;
```

Note how the values are set since the view parameters data structure (`vp`) is defined in the `main` routine and hence is not visible to the keyboard callback routine.

5.2 Navigation

We saw in Example 3 (page 26) how default mouse navigation was invoked using the function `mav_navigationMouse` (MFS p 174):

```
mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);
```

Related to this is the function `mav_navigationMouseDefaultParams` (MFS p 171):

```
void mav_navigationMouseDefaultParams(MAV_window *w, int but,
                                      MAV_navigatorFn x, float xls, float xas,
                                      MAV_navigatorFn y, float yls, float yas);
```

which controls the default mouse navigation.

`w` and `but` respectively specify the window and mouse button. The remaining arguments are two sets of three values, the first set for horizontal (or x) mouse movement, the second for vertical (or y). The three arguments in each set are:

- `x` (and `y`): a navigator function to perform the type of navigation required (described below).
- `xls` (and `yls`): the scaling factor to convert from pixels into application units in order to apply linear movements.
- `xas` (and `yas`): the scaling factor to convert from pixels into radians in order to apply angular movements (this is usually independent of the application).

For example, the following defines navigation triggered by the left mouse button. Horizontal movements of the mouse yaws the view; and vertical movement moves the view forward:

```
mav_navigationMouseDefaultParams(mav_win_all, MAV_LEFT_BUTTON,
                                 mav_navigateYaw, 0.02, -0.001,
                                 mav_navigateForwards, 0.02, 0.001);
```

A vertical mouse movement of 100 pixels equates to the eyepoint moving 2 (100×0.02) application units forwards.

Note that `as` is negative for `mav_navigateYaw`. This is because a right-handed coordinate system is assumed which implies that a positive yaw will rotate the view to the left. This is the opposite to what is required, and so a negative scaling factor is used to compensate.

Section A.3 describes how the linear scaling factor can be changed at run-time.

5.2.1 The default navigator functions

This section lists all the default navigator functions in MAVERIK. See Chapter 8 (page 91) for details of how to create your own customised navigators.

The scaling factor applicable to each navigator function is shown in brackets at the end of the description. All the navigator functions implemented so far use either `ls` or `as` but never both (however, it is conceivable that a navigator function could be written which does).

- `mav_navigateNull` does nothing.
- `mav_navigateTransX` translates the eyepoint along the world x-axis (`ls`).
- `mav_navigateTransY` translates the eyepoint along the world y-axis (`ls`).
- `mav_navigateTransZ` translates the eyepoint along the world z-axis (`ls`).

- mav_navigateRotRight rotates the view direction vectors and the eyepoint about the view right vector (as).
- mav_navigateRotUp rotates the view direction vectors and the eyepoint about the view up vector (as).
- mav_navigateForwards moves the eyepoint forwards along the view direction vector (ls).
- mav_navigateForwardsFixedUp moves the eyepoint along the projection of the view vector onto the plane normal to the global up vector (ls).
- mav_navigateUp moves the eyepoint along the view up vector (ls).
- mav_navigateUpFixedUp moves the eyepoint along the global up vector (ls).
- mav_navigateRight moves the eyepoint along the view right vector (ls).
- mav_navigateRightFixedUp moves the eyepoint along the projection of the view right vector onto the plane normal to the global up vector (ls).
- mav_navigateRoll rotates the view vectors about the view direction vector (as).
- mav_navigatePitch rotates the view vectors about the view right vector (as).
- mav_navigatePitchFixedUp rotates the view vectors about the projection of the view right vector onto the plane normal to the global up vector (as).
- mav_navigateYaw rotates the view vectors about the view up vector (as).
- mav_navigateYawFixedUp rotates the view vectors about the world up vector by an amount (as).

The default mouse navigation parameters are internally set within MAVERIK with the calls:

```
mav_navigationMouseDefaultParams(mav_win_all, MAV_LEFT_BUTTON,
                                 mav_navigateYawFixedUp, 0.01, -0.0005,
                                 mav_navigateForwardsFixedUp, 0.01, 0.0005);

mav_navigationMouseDefaultParams(mav_win_all, MAV_RIGHT_BUTTON,
                                 mav_navigateRight, 0.01, 0.0005,
                                 mav_navigateUp, 0.01, 0.0005);
```

The behaviour of the default mouse navigation is modified in Example 8 by the call:

```
mav_navigationMouseDefaultParams(mav_win_all, MAV_RIGHT_BUTTON,
                                 mav_navigateYawFixedUp, 0.01, -0.0005,
                                 mav_navigatePitchFixedUp, 0.01, 0.0005);
```

so that the right mouse button yaws and pitches the view.

5.2.2 Keyboard navigation

Example 8 also uses keyboard navigation, the interface to which is very similar to that of mouse navigation. Keyboard navigation is invoked using `mav_navigationKeyboard` (MFS p 170):

```
mav_navigationKeyboard(mav_win_all, mav_navigationKeyboardDefault);
```

The default keyboard navigation gives you the following "Doom" style controls:

- Cursor keys – navigate forwards/backwards and yaw;
- Page up/down – navigate up/down;
- Alt-Cursor left/right – sidestep left/right;
- Alt-Page up/down – pitch view up/down;
- Holding down "shift" doubles the rate of movement.

Control of the default keyboard navigation is more limited that the mouse variety since you can't redefine the actions taken by the various keys.

The function `mav_navigationKeyboardDefaultParams` (MFS p 167):

```
void mav_navigationKeyboardDefaultParams(MAV_window *w, float am, float ls, float as);
```

allows you to define the linear and angular scaling factors (`ls` and `as`) used by the default keyboard navigation. The value `am` can be thought of as the amount of movement a key gives (this value is multiplied by the appropriate scaling factor to give the true movement). Setting `am` to 50 (its default value) makes a navigation function invoked by the keyboard equivalent to it being invoked by 50 pixels of mouse movement.

5.3 User-defined data

Example 8 demonstrates a method of associating application-specific data structures with the standard MAVERIK object.

Recall that each MAVERIK object has a void pointer field, `userdef`, in its data structure. As the name suggests, this is a user-definable field that an application is free to use as it wishes – MAVERIK never interprets it. Typically, an application will use this field to point to its own data structures.

In this example a unique number is associated with each box in the scene as follows:

```
/* Application specific data structure */
typedef struct {
  int no;
} MyStruct;

/* Define a box */
void defBox(MAV_box *b, int no)
{
  /* Create and fill in application specific data structure */
  MyStruct *ms= (MyStruct *) mav_malloc(sizeof(MyStruct));
  ms->no= no;

  /* code omitted */

  /* Set userdef part to point to application-specific data structure */
  b->userdef= ms;
}
```

The boxes are then created in `main` with:

```
defBox(&box[i], i);
```

where `i` is the loop counter.

This number is obtained in the keyboard event callback trap which increases the size of the box (the ‘b’ key):

```
/* Convert from generic Maverik object to a box object */
MAV_box *box= (MAV_box *) mav_objectDataGet(ke->obj);

/* Get application specific data structure */
MyStruct *ms= (MyStruct *) box->userdef;

/* Increase size of box */
box->size.x+=0.5;

/* Print contents of application specific data structure */
printf("box number %i\n", ms->no);
```

Chapter 6

Miscellaneous Level 1 topics

We've now come to the end of the worked examples for the MAVERIK Programming Level 1 section of this manual. This chapter describes various miscellaneous concepts and function calls that a Level 1 programmer will need to know about, but which have not been covered by the worked examples. For example: changing the background colour; defining materials; opening multiple windows; and stereo viewing. Some of these ideas and functions are demonstrated by undocumented examples which can be found in the various sub-directories of the `examples/misc` directory of the MAVERIK distribution.

This chapter should be considered as an appendix or reference section to Programming Level 1. Some repetition of the material in the earlier chapters is inevitable.

6.1 Rendering

In this section we describe the MAVERIK functions and datatypes used to control object rendering. The MAVERIK rendering model is very similar to the OpenGL rendering model, and we assume that the reader is familiar with the OpenGL approach (if not, please refer to OpenGL documentation).

6.1.1 Rendering palettes

The way in which an object is rendered is controlled by its "surface parameters", which index into a **rendering palette**. By default, MAVERIK uses a single rendering palette, called `mav_palette_default`, which is created and initialised by `mav_initialise`.

A palette contains an ambient light specification, a **light table**, a **colour table**, a **materials table**, a **texture table** and a **font table**.

Each MAVERIK window is associated with a palette, and when the window is first created, it automatically becomes associated with `mav_palette_default`.

An application can define a new palette using `mav_paletteNew` (MFS p 279):

```
MAV_palette *mav_paletteNew(void);
```

and can associate a palette `p` with a window `w` using `mav_windowPaletteSet` (MFS p 282):

```
void mav_windowPaletteSet(MAV_window *w, MAV_palette *p);
```

Upon creation the contents of the palette are undefined. A palette supports a maximum of `mav_opt_maxColours` colours, `mav_opt_maxMaterials` materials, `mav_opt_maxTextures` textures, `mav_opt_maxFonts` fonts and `mav_opt_maxLights` lights. The default for these values, 150, 150, 150, 10 and 5 respectively, can be changed before MAVERIK is initialised but they must not be modified afterwards.

6.1.2 Surface parameters

An object's surface parameters refer to entries in the palette associated with the window in which the object is rendered. Surface parameters are encoded in the `MAV_surfaceParams` (MFS p 29) data structure. Each object stores a pointer to a `MAV_surfaceParams` data structure, which allows several objects to share a common set of surface parameters:

```
typedef struct {
  int mode;     /* rendering mode */
  int colour;   /* emissive colour */
  int material; /* ambient, diffuse and specular material */
  int texture;  /* texture map */
} MAV_surfaceParams;
```

The way the fields are interpreted depends on `mode`, as follows:

`mode` = `MAV_COLOUR`: the object has a uniform emissive colour, obtained from palette colour table entry `colour`.

`mode` = `MAV_MATERIAL`: the object has ambient, diffuse, specular and emissive surface material properties, obtained from palette material table entry `material`.

`mode` = `MAV_TEXTURE`: the object has a texture mapped onto its surface, obtained from palette texture table entry `texture`.

`mode` = `MAV_LIT_TEXTURE`: the object has a texture mapped onto its surface, which is also lit as specified for `MAV_MATERIAL` mode. Both the palette material table entry `material`, and the palette texture table entry `texture` are used.

`mode` = `MAV_BLENDED_TEXTURE`: the object has a colour which is an interpolation between the material and texture colours, governed by the alpha value of the texture. When alpha = 0, colour= material; when alpha = 1, colour = texture. Both the palette material table entry `material`, and the palette texture table entry `texture` are used.

An example program showing the different texture rendering modes can be found in the `textures` sub-directory of the miscellaneous examples.

To create a new set of surface parameters, use `mav_surfaceParamsNew` (MFS p 207), which for convenience also sets the initial values:

```
MAV_surfaceParams *mav_surfaceParamsNew(int mode, int col, int mat, int texture);
```

For example,

```
box.sp= mav_surfaceParamsNew(MAV_COLOUR, 5, 0, 0);
```

creates and assigns to `box.sp` a new set of surface parameters which specify that the object is to be rendered using `MAV_COLOUR` mode, using palette colour index 5. In this example, the irrelevant material table and texture table indexes are set to 0.

6.1.3 Defining colours, materials and textures

An emissive colour is defined using four floats, each in the range 0–1, for the red, green, blue and alpha (RGBA) components, using `mav_paletteColourSet` (MFS p 183):

```
void mav_paletteColourSet(MAV_palette *p, int index, float r, float g, float b, float a);
```

The alpha component is used to define the “transparency” of a colour, i.e. how much of the underlying RGB colour is visible through the defined RGB colour. An alpha value of zero makes the colour fully transparent, a value of one makes it fully opaque and fractional values cause the resultant colour to be a blending of the underlying colour with the defined colour. N.B. In order for transparency to work correctly the option variable `mav_opt_trans` must be set to `MAV_TRUE` before MAVERIK is initialised.

A material has four sets of RGBA values for the ambient, diffuse, specular and emissive components plus a value for its “shininess”, set using `mav_paletteMaterialSet` (MFS p 189):

```
void mav_paletteMaterialSet(MAV_palette *p, int index,
                            float ar, float ag, float ab, float aa,
                            float dr, float dg, float db, float da,
                            float sr, float sg, float sb, float sa,
                            float er, float eg, float eb, float ea,
                            float shin);
```

Texture maps are defined from file using `mav_paletteTextureSet` (MFS p 197):

```
int mav_paletteTextureSet(MAV_palette *p, int index, char *filename);
```

MAVERIK itself only supports the PPM (raw or ASCII encodings) and PNG[1] file formats for textures. However, MAVERIK can use the ImageMagick `convert` program, if installed, to convert almost any other image file format into PPM or PNG and then parse that. This conversion process is hidden from the user so that MAVERIK appears to support virtually all image file formats. Furthermore, MAVERIK uses ImageMagick's `convert` program to resize the image, if needed, so that it is an integer power of 2 in both width and height – a requirement placed on texture images by OpenGL.

6.1.4 Texture manipulation

`mav_paletteTextureSetFromMem` (MFS p 196) defines a texture map from an area of memory rather than a file, to allow the procedural generatation of textures:

```
int mav_paletteTextureSetFromMem(MAV_palette *p, int index, int width, int height,
                                 unsigned long *mem);
```

where `mem` is ABGR ordered.

The mipmapping of textures is controlled by the global option variable `mav_opt_mipmapping` (which is `MAV_FALSE` by default) and by the function `mav_paletteTextureMipmappingSet` (MFS p 195) which overrides the global default for a specific texture:

```
void mav_paletteTextureMipmappingSet(MAV_palette *p, int index, int v);
```

where `v` is set to either `MAV_TRUE` or `MAV_FALSE` to enable or disable mipmapping respectively. If a texture is to be mipmapped then this must be specified *before* it is defined with `mav_paletteTextureSet` or `mav_paletteTextureSetFromMem`. Once defined with mipmapping enabled, this function can be used to specify if a texture is mipmapped when rendered. An example of mipmapping can be found in the `textures` sub-directory of the miscellaneous examples .

`mav_paletteTextureAlphaSet` (MFS p 191) sets the alpha component of a texture to be some value. This allows for transparent textures:

```
void mav_paletteTextureAlphaSet(MAV_palette *p, int index, float a);
```

`mav_paletteTextureColourAlphaSet` (MFS p 192) is similar to the above but only sets the alpha for pixels whose colour is `r,g,b`:

[1] PNG support must be specified when MAVERIK is compiled and the PNG and zlib libraries must be installed.

```
void mav_paletteTextureColourAlphaSet(MAV_palette *p, int index,
                                      int r, int g, int b, float a);
```

This gives textures, portions of which are transparent. Since `r,g,b` are used in a comparison test they are ints in the range 0–255 rather than floats.

Typically you would make a texture with the portions you wish to be transparent to be, say, black and then use this function to set the alpha component of that colour.

A texture environment is a set of parameters governing how textures are applied. Specially it covers how minification and magnification of the texture is to be performed and if texture coordinates are clamped or repeated. These are separate issues from whether the texture is applied decal or modulating the underlying colour.

Rather than try to cater for all the possible combinations, MAVERIK relies on the application to define a callback function to set the relevant texture environment for the texture. The callback function is called each time a texture is rendered. `mav_paletteTextureEnvPaletteSet` (MFS p 193) and `mav_paletteTextureEnvSet` (MFS p 334) sets this callback function on a per-palette and per-texture basis.

```
void mav_paletteTextureEnvPaletteSet(MAV_palette *p, MAV_texEnvFn fn);
int mav_paletteTextureEnvSet(MAV_palette *p, int index, MAV_texEnvFn fn);
```

A callback set for a texture takes precedence over one defined for a palette. When a palette is created it has a default per-palette callback defined for it and when textures are created they have no per-texture callback defined. The default per-palette callback implements the common texture environment of repeating texture coordinates and using linear interpolation to perform minification and magnification.

6.1.5 Defining fonts

A font is defined using the function `mav_paletteFontSet` (MFS p 184):

```
void mav_paletteFontSet(MAV_palette *p, int index, char *s);
```

where `p` and `index` have the usual meaning and `s` is a string defining the X font to use. X fonts have a cryptic, but logical, naming scheme. Look at the `fonts` sub-directory of the miscellaneous examples to see this function in action. The names of the X fonts available on your machine can be found with the standard `xfontsel` program.

6.1.6 Defining lights

Material definitions only make sense if the scene is lit. MAVERIK provides the following default light and lighting model:

- Lighting model: RGBA (0.4, 0.4, 0.4, 1.0) using a local viewer;
- Light: ambient (0,0,0,1), diffuse (1,1,1,1), specular (1,1,1,1), positioned at (100, 150, 150).

An application can redefine the lighting model using `mav_paletteLightingModelSet` (MFS p 188):

```
void mav_paletteLightingModelSet(MAV_palette *p, float ar, float ag, float ab, float aa,
                                 int local);
```

which takes as its parameters the ambient RGBA value for the scene and an indication of whether to use local, as opposed to infinite, viewer lighting calculations.

The definition of a light source specifies RGBA values for the ambient, specular and diffuse components, via `mav_paletteLightSet` (MFS p 187):

```
void mav_paletteLightSet(MAV_palette *p, int index,
                         float ar, float ag, float ab, float aa,
                         float dr, float dg, float db, float da,
                         float sr, float sg, float sb, float sa);
```

Lights are positioned using a vector to define their location using `mav_paletteLightPos` (MFS p 185):

```
  void mav_paletteLightPos(MAV_palette *p, int index, MAV_vector pos);
```

The function `mav_paletteLightPositioning` (MFS p 186) defines whether this position is relative to the eye point or is in world coordinates:

```
  void mav_paletteLightPositioning(MAV_palette *p, int index, int pos);
```

If `pos` is set to `MAV_LIGHT_RELATIVE` (the default) the position is relative to the eye point, and subsequently follows it, to give a car-headlight effect. Setting `pos` to `MAV_LIGHT_ABSOLUTE` specifies the position is in world coordinates to give the effect of a light at a fixed position in the model.

An example of positioning lights can be found in the `lights` sub-directory of the miscellaneous examples.

6.1.7 Finding an empty or matching palette index

Functions exist for obtaining an empty (i.e. unused) index for the various components in a palette. For example, `mav_paletteColourIndexEmptyGet` (MFS p 180):

```
int mav_paletteColourIndexEmptyGet(MAV_palette *p);
```

returns an empty colour index in the supplied palette. If no empty index can be found, -1 is returned and a warning message printed to `stderr`.

Similar functions exist, described in the MFS on the same page as the above function, for returning empty indices for the other components of a palette.

A matching colour index in a palette can be found by using `mav_paletteColourIndexMatchGet` (MFS p 181):

```
int mav_paletteColourIndexMatchGet(MAV_palette *p, float r, float g, float b, float a);
```

which returns the index of a colour which matches the supplied values, or -1 if no match can be found. As above, similar functions exist for the other components of a palette.

6.2 Windows

6.2.1 Specifying a perspective view

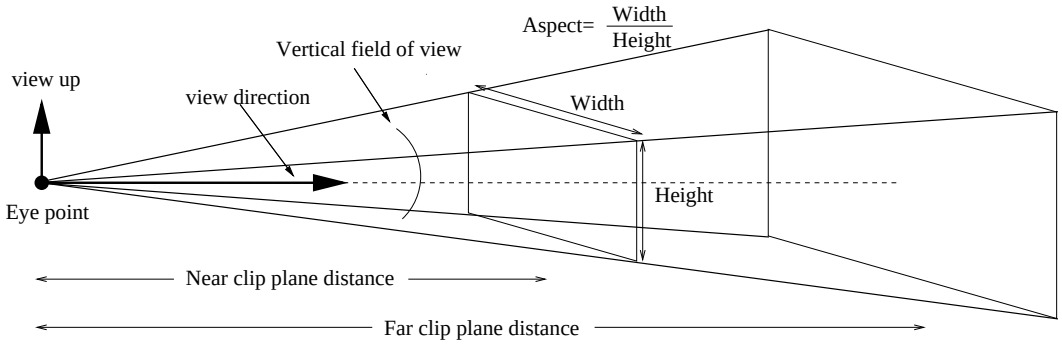
So far, we have discussed the viewing parameters which describe the application's view of the virtual environment. In order to display this view in a window we also need to define how that view is projected onto the screen to give the final scene. Using the standard analogy of a camera, we not only need to position the camera but to also decide which type of lens to use.

In common with most graphics systems, MAVERIK makes a distinction between the processes of defining the view and defining how that is projected onto the screen. The most common type of projection is a perspective projection, which is specified using `mav_windowPerspectiveSet` (MFS p 227):

```
void mav_windowPerspectiveSet(MAV_window *w, float ncp, float fcp, float fov,
                              float aspect);
```

This function defines a perspective projection for window `w` with near clip plane distance `ncp`, far clip plane distance `fcp`, a vertical field of view `fov` and aspect ratio `aspect`. `fov` is defined in degrees and is in the range [0–180]. To give a distortion-free projection, `aspect` should match the aspect ratio of the window.

The figure below shows how these parameters are used (please refer to the standard OpenGL documentation if you are unfamiliar with the terms).



The default values are:

```
ncp= 0.1;
fcp= 1000.0;
fov= 60;
aspect= same as the window
```

Section A.3 describes how the perspective view parameters can be changed at run-time.

6.2.2 Specifying an orthogonal view

An orthogonal projection is specified using `mav_windowOrthogonalSet` (MFS p 226):

```
void mav_windowOrthogonalSet(MAV_window *w, float ncp, float fcp, float size,
                             float aspect);
```

`w`, `ncp`, `fcp` and `aspect` are described above. `size` is the vertical extent, in application units, of the orthogonal projection.

6.2.3 Stereo viewing

Hardware

There are basically two ways of achieving stereo output both of which are supported by MAVERIK. The first is to create two separate windows, one for the left eye view the other for the right eye view, and employ some hardware, such as SGI's multi-channel option, to generate separate video signals for each window. The other method is to use a stereo graphics context, more commonly known as quad-buffer stereo. Modern LCD shutter glasses typically use quad-buffer stereo (the older interlaced type of shutter glasses can be supported by using two separate windows carefully positioned such that they completely overlap when the monitor is in stereo mode).

Software

The first step to achieve stereo output is to request a pair of windows be opened by `mav_initialise`, one for the left eye view the other for the right eye view (while for quad-buffer stereo only one window is actually opened, conceptually the left and right buffers/views are addressed separately). This is achieved by setting the MAVERIK global variable `mav_opt_stereo` (see Section C.1.3, page 158) before the initialisation call. Acceptable values are `MAV_STEREO_TWO_WINS` and `MAV_STEREO_QUAD_BUFFERS`:

```
mav_opt_stereo= MAV_STEREO_TWO_WINS;
mav_initialise(&argc, argv);
```

The two views are associated with a single set of stereo parameters, which by default is `mav_stp_default`, the contents of which defines the offset between the left and right eye views. The default implementation of stereo viewing, as described here, produces two parallel views of the virtual environment offset by some amount. There is no view convergence. We will discuss in Chapter 11.2 (page 114) how users can define their own methods of stereo viewing.

The stereo offset is set as follows:

```
/* Define stereo parameters, i.e. the stereo offset */
mav_stp_default.offset= 0.5;
```

The eye point in the left and right views is offset from its original position along the view right vector by an amount $-\text{offset}/2.0$ and $+\text{offset}/2.0$ respectively. Unlike navigation, this offset is only temporary and does not affect the values stored in the view parameters.

MAVERIK provides some keyboard function keys for fine-tuning stereo views at run-time. These are described in Section A.3 (page 126).

An example of stereo viewing can be found in the `stereo` sub-directory of the miscellaneous examples.

Note that in a quad-buffer setup there is usually only one depth buffer which is shared by the two views. MAVERIK clears the depth buffer as each view becomes active for rendering. This means that all of one view must be rendered before all of the other view – you cant swap between them more than once per frame. Some machines support separate depth buffers and this can be indicated by setting `mav_opt_stereo` to be `MAV_STEREO_QUAD_BUFFERS_SEPARATE_Z`.

6.2.4 Background colour

When a new window is created, its background colour is set by default to RGB value (0.0, 0.5, 1.0). This can be changed using `mav_windowBackgroundColourSet` (MFS p 217):

```
void mav_windowBackgroundColourSet(MAV_window *w, float r, float g, float b);
```

`w` is the window to set, and `r`, `g` and `b` specify the required RGB background colour.

6.2.5 Backface culling

By default, backface culling is disabled. It can be set using `mav_windowBackfaceCullSet` (MFS p 216):

```
void mav_windowBackfaceCullSet(MAV_window *w, int v);
```

Setting `v` to `MAV_TRUE` enables backface culling; `MAV_FALSE` disables it. Vertices must be ordered anti-clockwise around the normal for backface culling to work.

6.2.6 Opening multiple windows

`mav_windowNew` (MFS p 225) opens a new window, and returns its handle:

```
MAV_window *mav_windowNew(int x, int y, int w, int h, char *name, char *disp);
```

where `x`, `y`, `w` and `h` specify the window's horizontal and vertical position on screen, and its width and height. `name` defines the window title that appears in the menu bar. `disp` is the name of the X display on which to open the window (setting the value to `NULL` uses the `DISPLAY` environment variable). Note, the window manager may not honor the requested parameters.

Windows are of X resource class "MaverikApp". This can be used to control various window attributes, such as the amount of window decoration and borders. For example, adding the line `4Dwm*MaverikApp*clientDecoration: none` to the `.Xdefaults` file will open a window without any decoration when using SGI's default window manager 4Dwm.

An example of opening multiple windows be found in the `windows` sub-directory of the miscellaneous examples.

6.2.7 Deleting windows

`mav_windowDelete` (MFS p 219) deletes a window:

```
void mav_windowDelete(MAV_window *w);
```

N.B. You can not delete the first window opened, only subsequently opened windows, since the initial window creates data which other windows share.

An example of deleting windows be found in the `windows` sub-directory of the miscellaneous examples.

Part III

MAVERIK Programming Level 2

Chapter 7

Creating new classes of object

As we discussed in Chapter 2, the needs of, say, an application involved in architectural walkthroughs are quite different from one involved in abstract data visualization. Rather than trying to create a compromise system, our approach is to design a system that can be fully and easily customized so that the resultant virtual environment exhibits a behaviour which is customized to, and consistent with, the nature of the application.

This is a fundamental concept of MAVERIK and one which sets it apart from other VR systems. A key aspect in achieving this is the ability to create new object primitives tailored to the application.

For example, an application wishing to populate a virtual environment with particle systems which model smoke or fire, will probably want to represent this with its own object primitives which are defined and rendered in a particular way, rather than attempting to map its particles onto – say – the MAVERIK "sphere" object.

The objects MAVERIK provides such as box, sphere, polygon, and so on (see Appendix B for a full list), should be seen as (hopefully useful) defaults. Applications are under no obligation to use any of these objects.

An application can use **whatever data structure it likes** to represent an object. We use the term "object" as simply a convenient way of naming something which an application requires MAVERIK to treat as an entity. To take an example from a real project in which MAVERIK has been used extensively, an existing Computer-Aided Design application from the oil industry might wish to use a "pipe" primitive, which in addition to geometrical data, also contains non-geometric information such as the temperature and pressure of the liquid it carries.

To use such a data-rich primitive with a "traditional" VR system is very difficult. The pipe's geometrical data would typically have to be re-cast into a form dictated by the VR system – often polygon or solid primitive based. While the non-geometric information can be incorporated into "traditional" VR systems, it is usually in the form of "passive" data. Here, the data can not be exploited to affect, for example, the rendering or collision detection functions since these are buried deep in the system where the user has little or no access to them. Furthermore, importing application data into a VR system means there are two separate copies of the same underlying data. Keeping these two sets of

data synchronized is problematic.

With MAVERIK this is not the case – whatever data structure you choose to use to represent the pipe primitive can be used directly – **without modification** – by MAVERIK. For example, the data structure can be exactly the same as that required by an existing automated pipe routing algorithm that you wish to use to give your virtual environment a realistic behavior. Furthermore, since MAVERIK is a toolkit which is linked into your application, and not a separate executable, it can process the application data directly rather than requiring its own copy.

The user has full control of how the object is processed by MAVERIK, and so all of the information in the data structure can be used to customize the virtual environment to a specific application.

In this chapter, we describe how an application can define new classes of objects.

7.1 Example 9: creating a new class

An application defines a separate **class** for each kind of object it wishes MAVERIK to manipulate. It also defines a set of **methods** which operate on that class. Some of these methods are required by MAVERIK, if it is to be able to manage the object in a virtual environment – methods such as “draw object”; other methods will only ever be used by the application itself.

We begin with a simple generic example. Suppose an application uses the following data structure to represent an object, and creates an instance of it:

```
/* The data structure to represent the AppObject */
typedef struct {
  char *name;
} AppObject;

AppObject app_object; /* Create the instance */

app_object.name= "I'm an object";
```

To use this object with MAVERIK, the application first needs to call `mav_classNew` (MFS p 263) to create a new MAVERIK class to represent it:

```
MAV_class *AppObjectClass;

/* Create a new Maverik class to represent the object */
AppObjectClass= mav_classNew ();
```

The next step is to register the application object as a MAVERIK object of the class we’ve just created, as follows:

```
MAV_object *obj;

/* Register the AppObject as a Maverik object */
obj= mav_objectNew (AppObjectClass, &app_object);
```

This is exactly the same mechanism as when registering an object which is one the MAVERIK default object classes as seen in the previous examples. The `mav_objectNew` function takes two arguments: the name of the object's class (`AppObjectClass`), and a pointer to the data structure defining the object (`&app_object`). It's sometimes helpful to think of `mav_objectNew` as **binding** the object's data to the MAVERIK object instance.

7.2 Object methods

Every class of MAVERIK object has two components: a **data structure**, and a set of **methods** which operate on the data structure. This is analogous to the object-oriented paradigm of data and methods, however it is implemented in C using callback functions.

Normally, MAVERIK arranges to execute the methods (or callback functions) at an appropriate time – the "draw" method for a particular class of object, for example, will be executed when MAVERIK encounters an instance of that class of object in an SMS which is being processed for display.

The draw method for an object class, `MAV_callbackDrawFn` (MFS p 51), has the following prototype:

```
typedef int (*MAV_callbackDrawFn) (MAV_object *, MAV_drawInfo *);
```

This is how an application defines a draw method, which MAVERIK will execute every time it wishes to draw an `AppObject` object:

```
/* Draw an AppObject */
int AppObjectDraw (MAV_object *o, MAV_drawInfo *di)
{
  /* Convert from generic Maverik object to the AppObject object */
  AppObject *a= (AppObject *) mav_objectDataGet(o);

  /* This would normally be the graphics code to draw the object */
  printf("Drawing AppObject whose name is %s\n", a->name);

  return MAV_TRUE;
}
```

`AppObjectDraw` is executed with the MAVERIK object to draw, `o`, and a set of so-called "drawing information", `di`. The generic MAVERIK object, `o`, has to be converted into the `AppObject` object so the function can access its data to render it. This is achieved using the `mav_objectGetData` function

and is exactly the same process as seen with event-based callbacks for the "default" MAVERIK object classes.

The `MAV_drawInfo` (MFS p 73) data structure contains such information as the view clip planes and the eye point, and can be used to apply level of detail or fine culling on the object. We'll ignore this argument until Section 7.9.

The return value of the function indicates if the operation was successfully completed or not (for this simple example there is no reason why it should fail).

The draw callback is registered with MAVERIK using `mav_callbackDrawSet` (MFS p 260), as follows:

```
  /* Set the draw callback for the new class */
  mav_callbackDrawSet(mav_win_all, AppObjectClass, AppObjectDraw);
```

which sets the draw callback for a particular class of object to be a particular function. Note that this is set on a per-window basis so that the object could be rendered, for example, as wireframe in one window and as filled in another.

The full example, `eg9.c`, looks like this:

```
/* eg9.c */
#include "maverik.h"
#include <stdio.h>

/* The data structure to represent the AppObject */
typedef struct {
  char *name;
} AppObject;

/* Define an AppObject */
void defAppObject(AppObject *a)
{
  a->name= "I'm an object";
}

/* Draw an AppObject */
int AppObjectDraw(MAV_object *o, MAV_drawInfo *di)
{
  /* Convert from generic Maverik object to the AppObject object */
  AppObject *a= (AppObject *) mav_objectDataGet (o);

  /* No code for drawing the object -- this would normally be graphics! */
  printf("Drawing AppObject whose name is %s\n", a->name);

  return MAV_TRUE;
}
```

```
/* Render a frame */
void drawFrame(MAV_SMS *sms)
{
  /* Check for and act on any events */
  mav_eventsCheck();

  /* Request start of a new frame */
  mav_frameBegin();

  /* Display the SMS in all windows */
  mav_SMSDisplay(mav_win_all, sms);

  /* Request end of the frame */
  mav_frameEnd();
}

int main(int argc, char *argv[])
{
  MAV_class *AppObjectClass;
  AppObject app_object;
  MAV_object *obj;
  MAV_SMS *sms;

  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  /* Create a new Maverik class to represent the object */
  AppObjectClass= mav_classNew();

  /* Set the draw callback for the new class */
  mav_callbackDrawSet(mav_win_all, AppObjectClass, AppObjectDraw);

  /* Define an instance of the AppObject */
  defAppObject(&app_object);

  /* Register the AppObject as a Maverik object */
  obj= mav_objectNew(AppObjectClass, &app_object);

  /* Create an SMS and add the object to it */
  sms= mav_SMSObjListNew();
  mav_SMSObjectAdd(sms, obj);

  /* Rendering loop */
  while (1) drawFrame(sms);
}
```

Executing this example you should see the standard blue background with the message “Drawing AppObject whose name is I’m an object” scrolling up the shell window. This example is intended to show the general principles involved without getting bogged down in an actual implementation. We’ll now move on to show how a real 3D object would be created.

7.3 Example 10: the dodecahedron

To illustrate what's involved in defining the kind of realistic new object class that applications might wish to use, we present a worked example. We'll define a "dodecahedron" object – the fourth of the five platonic solids, comprising 12 faces, each of which is a pentagon.

We'll take as our starting point the default MAVERIK object class "box", whose data structure is as follows:

```
typedef struct {
  MAV_vector size;       /* size of box */
  MAV_surfaceParams *sp; /* surface parameters */
  MAV_matrix matrix;     /* transformation matrix */
  void *userdef;         /* user-defined data */
} MAV_box;
```

Our dodecahedron will be similar, but we'll specify its size using a single radius parameter:

```
typedef struct {
  float r;               /* size of dodecahedron */
  MAV_surfaceParams *sp; /* surface parameters */
  MAV_matrix matrix;     /* transformation matrix */
} MAV_dodec;
```

Note that we have retained the `sp` and `matrix` fields and that the name of the data structure starts with `MAV_`. We'll briefly describe why these choices were made.

7.3.1 Data structure choices

We've said that applications are under no obligation to use MAVERIK's data types, but in many cases this is the most convenient approach.

Most objects an application wishes to manage in a virtual environment will have a position/orientation and colour. Whatever is used to represent or calculate these, they ultimately have to be turned into calls which change the state of the underlying graphics system, for example OpenGL.

MAVERIK provides an easy and transparent means of achieving this, but at the cost of using its data types to represent these common object properties – `MAV_matrix` for coordinate transformations, and `MAV_surfaceParams` to control the "colour" which is used for rendering.

If an application is free to choose any data structure for object classes, then using the MAVERIK data types will make life a lot easier. If, on the other hand, it must represent objects using some fixed data structure, then while this is possible, it involves the application writing certain operations for itself.

We will return to this subject in Section 7.4 to show how the dodecahedron could be re-written so as not to use any MAVERIK data types.

7.3.2 Naming conventions

Naming the dodecahedron data structure `MAV_dodec` makes it appear to be part of the MAVERIK system and not an additional object created just for this application. In this section we explain the significance of this.

The MAVERIK system comprises a kernel and a number of so-called "supporting modules". At the kernel level there is surprisingly little – no objects, no input devices, no navigation, not even the concept of rendering an object. What the kernel does provide, however, is a framework in which these can be defined.

The supporting modules use this framework to provide the support for building applications. For example, one module provides mouse and keyboard input; another provides the ability to define a rendering callback; another defines the default graphical primitive classes, and so on.

Therefore what we have until now called "default" MAVERIK objects are not part of the MAVERIK kernel. They have been added afterwards in just the same manner as shown in `eg9` for the generic object and we are about to see for the dodecahedron.

Furthermore, the supporting modules are extensible. If the dodecahedron object turns out to be useful, it can be encapsulated in a supporting module and added to the MAVERIK system (we will show how this is achieved in Chapter 12). It would then appear to another user just as much a part of MAVERIK as the `MAV_box` or `MAV_sphere` are. With this in mind it makes sense to use a consistent naming scheme when defining new objects. Of course, this is not a requirement. If your object is so specific to a particular application that its of no possible use to anyone else, call it what you want. It's a fine point.

Back to the dodecahedron.

The following example, `eg10.c`, adds the dodecahedron object in the same manner as shown for the generic object in Example 9, but actually implements the rendering of this shape in the draw callback.

```
/* eg10.c */
#include "maverik.h"
#include <math.h>

/* The data structure and object class to represent the dodecahedron */
typedef struct {
  float r;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
} MAV_dodec;

MAV_class *mav_class_dodec;

/* The vertices of a unit sized dodecahedron */
```

```
#define V1 0.381966
#define V2 0.618034

MAV_vector vecs[]={{-V1,0,1},{V1,0,1},{-V2,-V2,-V2},{-V2,-V2,V2},{-V2,V2,-V2},{-V2,V2,V2},
 {V2,-V2,-V2},{V2,-V2,V2},{V2,V2,-V2},{V2,V2,V2},{1,V1,0},{1,-V1,0},{-1,V1,0},{-1,-V1,0},
 {-V1,0,-1},{V1,0,-1},{0,1,V1},{0,1,-V1},{0,-1,V1},{0,-1,-V1}};

/* Routine to render a pentagon given the vertices and size r */
void pentagon(int a, int b, int c, int d, int e, float r)
{
  MAV_vector v1, v2, norm;

  /* Calculate normal of pentagon from crossproduct of the 2 edges */
  v1= mav_vectorSub(vecs[a], vecs[b]);
  v2= mav_vectorSub(vecs[b], vecs[c]);
  norm= mav_vectorNormalize(mav_vectorCrossProduct(v1, v2));

  /* Render the pentagon as a polygon. Vecs contain unit pentagon, so mult by r */
  mav_gfxPolygonBegin();
  mav_gfxNormal(norm);
  mav_gfxVertex(mav_vectorScalar(vecs[a], r));
  mav_gfxVertex(mav_vectorScalar(vecs[b], r));
  mav_gfxVertex(mav_vectorScalar(vecs[c], r));
  mav_gfxVertex(mav_vectorScalar(vecs[d], r));
  mav_gfxVertex(mav_vectorScalar(vecs[e], r));
  mav_gfxPolygonEnd();
}

/* Routine to render the dodecahedron */
int mav_dodecDraw(MAV_object *o, MAV_drawInfo *di)
{
  MAV_dodec *dodec;

  /* Convert from generic Maverik object to the dodecahedron object */
  dodec= (MAV_dodec *) mav_objectDataGet(o);

  /* Set the correct colouring */
  mav_surfaceParamsUse(dodec->sp);

  /* Store the current transformation matrix then multiply it by the local transformation */
  mav_gfxMatrixPush();
  mav_gfxMatrixMult(dodec->matrix);

  /* Render the 12 pentagons that make up the dodecahedron */
  pentagon(0, 1, 9, 16, 5, dodec->r);
  pentagon(1, 0, 3, 18, 7, dodec->r);
  pentagon(1, 7, 11, 10, 9, dodec->r);
  pentagon(11, 7, 18, 19, 6, dodec->r);
  pentagon(8, 17, 16, 9, 10, dodec->r);
  pentagon(2, 14, 15, 6, 19, dodec->r);
  pentagon(2, 13, 12, 4, 14, dodec->r);
  pentagon(2, 19, 18, 3, 13, dodec->r);
  pentagon(3, 0, 5, 12, 13, dodec->r);
```

```
  pentagon(6, 15, 8, 10, 11, dodec->r);
  pentagon(4, 17, 8, 15, 14, dodec->r);
  pentagon(4, 12, 5, 16, 17, dodec->r);

  /* Restore original transformation matrix */
  mav_gfxMatrixPop();

  return MAV_TRUE;
}

/* Define a dodecahedron */
void defDodec(MAV_dodec *d)
{
  d->r=2.5;
  d->sp= mav_sp_default;
  d->matrix= MAV_ID_MATRIX;
}

/* Render a frame */
void drawFrame(MAV_SMS *sms)
{
  /* Check for and act on any events */
  mav_eventsCheck();

  /* Request start of a new frame */
  mav_frameBegin();

  /* Display the SMS in all windows */
  mav_SMSDisplay(mav_win_all, sms);

  /* Request end of the frame */
  mav_frameEnd();
}

int main(int argc, char *argv[])
{
  MAV_dodec dodec;
  MAV_object *obj;
  MAV_SMS *sms;
  float r=0;

  /* Initialise the Maverik system */
  mav_initialise(&argc, argv);

  /* Create a new class to represent the dodecahedron */
  mav_class_dodec= mav_classNew();

  /* Set the draw callback for this new class */
  mav_callbackDrawSet(mav_win_all, mav_class_dodec, mav_dodecDraw);

  /* Define a dodecahedron */
  defDodec(&dodec);
```

```
  /* Register the dodecahedron as a Maverik object */
  obj= mav_objectNew(mav_class_dodec, &dodec);

  /* Create a SMS */
  sms= mav_SMSObjListNew();

  /* Add object to SMS */
  mav_SMSObjectAdd(sms, obj);

  /* Use default mouse navigation */
  mav_navigationMouse(mav_win_all, mav_navigationMouseDefault);

  /* Rendering loop */
  while (1) {
    /* Spin the dodecahedron */
    r+=1;
    dodec.matrix= mav_matrixSet(r,r*2,r/2, 0,0,0);

    /* Draw a frame */
    drawFrame(sms);
  }
}
```

Running this example should show a tumbling red dodecahedron. The tumbling effect is achieved by setting the object's transformation matrix at each frame with the function `mav_matrixSet`. This function returns a transformation matrix defined by a roll, pitch, yaw orientation (the first 3 arguments in degrees) and (x, y, z) position (the remaining 3 arguments). Note: roll, pitch and yaw are arbitrarily chosen to be rotations about the Z, X and Y axis respectively.

We'll now look at the draw callback function, `mav_dodecDraw`, in more detail. The first action performed with the dodecahedron data structure is to use the correct set of surface parameters, achieved with:

```
  /* Set the correct colouring */
  mav_surfaceParamsUse(dodec->sp);
```

This sets the underlying graphics system to be in a correct state to render the object as desired, such as enabling lighting, disabling texturing, and so on.

The next action is then to render the shape. In common with many graphics systems, MAVERIK uses the notion of a modelview matrix which can be pushed, popped, set and multiplied to change between coordinate frames. This idea should be familiar to anyone with a working knowledge of OpenGL (the level of reader we are assuming).

```
  /* Store the current transformation matrix -
     then multiply it by the local transformation */
  mav_gfxMatrixPush();
  mav_gfxMatrixMult(dodec->matrix);
```

```
/* Render the 12 pentagons that make up the dodecahedron */
[code removed]

/* Restore original transformation matrix */
mav_gfxMatrixPop();
```

7.3.3 Abstracted graphics layer

Any function that starts with `mav_gfx` is part of MAVERIK's abstracted graphics layer. In reality this is little more than a wrapper to OpenGL. For the calls above:

`mav_gfxMatrixPush` (MFS p 267) is equivalent to `glPushMatrix`
`mav_gfxMatrixMult` (MFS p 267) is equivalent to `glMultMatrixf`
`mav_gfxMatrixPop` (MFS p 267) is equivalent to `glPopMatrix`

MAVERIK uses this abstracted graphics layer to hide the specifics of the underlying graphics systems thus allowing application source code to be simply re-linked, not re-written, to make use of different graphics systems.

The default MAVERIK library resolves the abstracted graphics layer into their OpenGL equivalents. However, MAVERIK can be configured to resolves this layer into DirectX calls or, although now unsupported, IrisGL calls. It is even possible to configure MAVERIK to resolve the layer without making any graphical calls at all! (which could be used in making an offline application such as a raytracer which used MAVERIK's object and spatial management algorithms).

It is hoped that this mechanism could be used to support other immediate mode rendering systems of similar specification to OpenGL.

That said, there is nothing stopping the user from placing direct OpenGL calls in the rendering call-back functions. Obviously, doing this would then rule out the possibility of using the other graphics systems described above.

The 20 vertices which make up a unit-sized dodecahedron are stored as an array of vectors. Twelve pentagons can be defined, using a combination of these vertices, to render the dodecahedron. Before rendering each vertex is multiplied by the dodecahedron's radius so that it has the correct size. (The rendering of this shape was borrowed, with gratitude, from the GLUT source code.)

Note the use of MAVERIK's abstracted graphics layer:

`mav_gfxPolygonBegin` (MFS p 268) is equivalent to `glBegin(GL_POLYGON)`
`mav_gfxNormal` (MFS p 268) is equivalent to `glNormal3f`
`mav_gfxVertex` (MFS p 268) is equivalent to `glVertex3f`
`mav_gfxPolygonEnd` (MFS p 268) is equivalent to `glEnd`

As with any other graphics system a surface normal must be defined for the pentagon so as it will appear correctly lit. This is trivially calculated as the cross product of two edges.

7.4 Application independence

In the dodecahedron example, we used MAVERIK types for some of the object data fields:

```
MAV_surfaceParams *sp;
MAV_matrix matrix;
```

As mentioned previously, an application is under no obligation whatsoever to use MAVERIK data types (although life is lot simpler if it does).

For example, instead of using a `MAV_matrix` to define an object's position and orientation we could use `x`, `y` and `z`, and `roll`, `pitch` and `yaw` to define it.

In the rendering callback we would then need to replace the line:

```
mav_gfxMatrixMult(dodec->matrix);
```

with

```
mav_gfxMatrixMult(mav_matrixSet(dodec->roll, dodec->pitch, dodec->yaw,
                                dodec->x, dodec->y, dodec->z));
```

where the function `mav_matrixSet` defines a transformation matrix with that orientation and position. The principle here is that whatever form the position/orientation data takes it must be converted into, but not necessarily stored in the data structure as, a `MAV_matrix` in order for it to be applied to the underlying graphics system.

Since Euler angles are a popular means of defining orientation, MAVERIK provides support for converting them into a `MAV_matrix`. However, if your orientation was in a more exotic form (the Euler angles being represented by integers in the range 0–255, or by using quaternions, for example) then you have to perform the mathematics of the conversion into a `MAV_matrix` yourself.

Similarly, an object's colour could, for example, be defined using a single colour index rather than a MAVERIK surface parameters datatype. Again, regardless of how the colour is stored it can be converted into a MAVERIK surface parameters data type in the rendering callback function and applied in the usual manner. For example:

```
MAV_surfaceParams sp;

sp.mode= MAV_COLOUR;
sp.colour= dodec->colIndex;
mav_surfaceParamsUse(&sp);
```

where the appropriate RGB values for the colour are defined elsewhere. You do not necessarily have to use the `mav_surfaceParamsUse` (MFS p 281) function to define the colouring scheme of an object. You can, if you wish, make direct calls to OpenGL to set the correct state. However, if you do this you must call the function `mav_surfaceParamsUndefine` (MFS p 280) so as to notify MAVERIK that its internal notion of the state of the graphics system is no longer valid.

7.5 Example 11: the “bounding box” method

The bounding box (BB) method computes an axis-aligned bounding box for an object taking into account the object’s transformation, but not any additional transformations that may have been applied – only one level of transformation is taken into account in the calculation. Put another way, in the examples so far the objects would return a world coordinate frame BB whereas if they were the sub-object in a hierarchical structure then they would return a BB in the coordinate frame of the **parent object** which would then transform this into the world coordinate frame.

The BB of an object is used by MAVERIK to determine two things. First, if the object is within the view frustum and therefore should be displayed (that is, have its draw callback executed). Second, to determine which object the mouse was pointing at when a keyboard or mouse event occurred. Note that if the BB computation is a “bad fit” to the actual shape of the object, MAVERIK may incorrectly report selections since the mouse may be pointing outside of the object but could still be within its BB. We will show in the next section how to perform accurate object selection.

The BB method for an object, `MAV_callbackBBFn` (MFS p 48), has the following prototype:

```
typedef int (*MAV_callbackBBFn) (MAV_object *, MAV_BB *);
```

The callback function takes as its first argument the MAVERIK object to be processed and returns the calculated BB in the second argument. The `MAV_BB` (MFS p 5) datatype comprises two `MAV_vectors`, `min` and `max`, to define the BB’s extent. As with the draw callback, the return value indicates the success or failure of the operation.

The dodecahedron BB is calculated as follows:

```
int mav_dodecBB (MAV_object *o, MAV_BB *bb)
{
  MAV_dodec *dodec;
  MAV_BB local;

  /* Convert from generic Maverik object to the dodecahedron object */
  dodec= (MAV_dodec *) mav_objectDataGet (o);

  /* Local coordinate frame axis-aligned bounding box */
  local.min.x= -dodec->r;
  local.min.y= -dodec->r;
  local.min.z= -dodec->r;
```

```
  local.max.x= dodec->r;
  local.max.y= dodec->r;
  local.max.z= dodec->r;

  /* Align local coordinate frame with the parent (in this case the world) frame */
  mav_BBAlign (local, dodec->matrix, bb);

  return MAV_TRUE;
}
```

The BB of the dodecahedron in its local coordinate frame is a box centered at the origin and of extent twice the dodecahedron radius. The function `mav_BBAlign` (MFS p 233) calculates the axis-aligned BB in one coordinate frame, `bb`, given the BB defined in a different coordinate frame, `local`, and the transformation matrix between the two (the second argument). If you didn't store the dodecahedron's orientation and position as a MAVERIK matrix you would either have to convert it into one, or perform the maths to convert between the two coordinate frames yourself.

Obviously, this is just one way in which an axis-aligned BB can be calculated. Another method would be as follows:

```
int mav_dodecBB2 (MAV_object *o, MAV_BB *bb)
{
  MAV_dodec *dodec;
  int i;

  /* Convert from generic Maverik object to the dodecahedron object */
  dodec= (MAV_dodec *) mav_objectDataGet (o);

  /* Find BB enclosed by the points after size and position of */
  /* dodec have been accounted for */
  mav_BBCompInit (bb);
  for  (i=0; i<20; i++) {
    mav_BBCompPt (mav_vectorMult(mav_vectorScalar(vecs[i], dodec->r), dodec->matrix),
                  bb);
  }

  return MAV_TRUE;
}
```

In this implementation the 20 vertices which make up the dodecahedron are first multiplied by the scalar `dodec->r` to give a dodecahedron of the correct size, and then by the MAVERIK matrix `dodec->matrix` to give their position in the world coordinate frame.

Allied to this calculation are the functions `mav_BBCompInit` (MFS p 233) and `mav_BBCompPt` (MFS p 233). These functions are used when calculating a bounding box which comprises a collection of points. The second of these functions takes a vector and a pointer to a bounding box and modifies the contents of the bounding box so that it encompasses the vector. The first function simply initialises the contents of the BB.

These two implementations both perform the same job but have different advantages and disadvantages. The first method is quicker but overestimates the BB. Whereas the second method is slower but more accurate than the first.

Initially the first method is used and set using `mav_callbackBBSet` (MFS p 260) as follows:

```
/* Set the calculate BB callback for this new class */
mav_callbackBBSet (mav_win_all, mav_class_dodec, mav_dodecBB);
```

However, MAVERIK allows callback functions to be dynamically switched. In this example we define a keyboard event callback to switch the bounding box functions:

```
MAV_callbackBBFn fn;

if (ke->key=='b') { /* Toggle calc BB callback function */
  fn= (MAV_callbackBBFn) mav_callbackQuery (mav_callback_BB, mav_win_all, o);
  if (fn==mav_dodecBB)
  {
    mav_callbackBBSet (mav_win_all, mav_class_dodec, mav_dodecBB2);
  }
  else
  {
    mav_callbackBBSet (mav_win_all, mav_class_dodec, mav_dodecBB);
  }
}
```

The function `mav_callbackQuery` (MFS p 306) has the prototype:

```
MAV_callbackFn mav_callbackQuery (MAV_callback *cb, MAV_window *w, MAV_object *o);
```

and returns which callback function is set for callback `cb` in window `w` for the object `o`. This is returned as a generic callback function (`MAV_callbackFn` (MFS p 54)) and has to be cast to a BB callback function (`MAV_callbackBBFn`) in order for a comparison to be performed. The function returns `NULL` if no callback function has been set. A list of the callback handles for the different operations can be found in Section C.3 (the BB callback is identified with `mav_callback_BB`; the draw callback with `mav_callback_draw` etc...).

To show the effects of view frustum culling, a `printf` has been added to the draw callback in `eg11` to indicate when it's being called. In addition, the BB callback for the dodecahedron is explicitly called and the calculated BB displayed. This is achieved with the functions `mav_callbackBBExec` (MFS p 259) and `mav_BBDisplay` (MFS p 123) whose prototypes are:

```
int  mav_callbackBBExec (MAV_window *w, MAV_object *o, MAV_BB *bb);
void mav_BBDisplay (MAV_window *w, MAV_BB bb);
```

Execution of the BB callback is analogous to that described for the "get surface parameters" process-based callback in `eg7` (page 40). The second function simply renders, in black wireframe, the given BB in the given window.

Although not an issue in this example, suppose we added the dodecahedron to the SMS as follows:

```
mav_SMSObjectAdd (sms, mav_objectNew (mav_class_dodec, &dodec));
```

i.e. not noting the relationship between the application dodecahedron object and MAVERIK object. How then could we execute the BB callback on the dodecahedron since we don't know its corresponding MAVERIK object?

The reverse conversion (MAVERIK object to application object) is trivial, and performed by the function `mav_objectDataGet`. Conversion the other way (application object to MAVERIK object) is performed with the function `mav_objectDataWith` (MFS p 177) which takes as its only argument a `void` pointer to the application object data structure and returns the MAVERIK object which corresponds with that data, or `NULL` if the data has not been registered. The call to execute the BB callback in this example is peformed as follows to demonstrate this:

```
mav_callbackBBExec (mav_win_current, mav_objectDataWith (&dodec), &bb)
```

Running `eg11` shows the tumbling dodecahedron surrounded by its BB. Pressing 'b' over the dodecahedron toggles which implementation is used for calculating the BB, while 's' increases its size, and 'q' quits. Note how the messages printed to the shell window stop when the navigation takes the dodecahedron outside of the view frustum.

7.6 Example 12: the "intersection" method

Although a BB is enough for MAVERIK to perform rough-and-ready selection testing, a preferable approach is to define a method which accurately computes the intersection (if any) of an instance of the object class with a given vector.

The intersection method for an object, `MAV_callbackIntersectFn` (MFS p 59), has the following prototype:

```
typedef int (*MAV_callbackIntersectFn)(MAV_object *, MAV_line *,
                                       MAV_objectIntersection *);
```

The callback function takes as it arguments the MAVERIK object to process, the line with which to calculate the intersection with and a data structure in which to return the details of the intersection.

A `MAV_line` (MFS p 19) data structure consists of a two `MAV_vectors`: the origin of the line, `pt`, and normalized direction vector, `dir`, which are defined in the world coordinate frame. The `MAV_objectIntersection`

data structure contains a number of fields but only one is currently used, namely `pt1` – the distance from the line's origin to the first point of intersection with the object.

If the line does not intersect the object then `pt1` should be set to a negative value and the callback function returns `MAV_FALSE`. Conversely, it should return `MAV_TRUE` if the line does intersect the object. If the point originates inside the object, then the distance to closest intersection should be set to zero.

We define the intersection method for the dodecahedron as follows:

```
/* Function to calculate the object-line intersection of a pentagon */
void pentagonIntersect (MAV_line ln, MAV_objectIntersection *oi,
                        int a, int b, int c, int d, int e, float r)
{
  MAV_polygon apoly;
  MAV_vector v1, v2, norm;

  /* Calculate normal of pentagon from crossproduct of the 2 edges */
  v1= mav_vectorSub (vecs[a], vecs[b]);
  v2= mav_vectorSub (vecs[b], vecs[c]);
  norm= mav_vectorNormalize (mav_vectorCrossProduct (v1, v2));

  /* Make up a MAV_polygon to represent the pentagon */
  apoly.np= 5;
  apoly.norm= norm;
  apoly.vert= mav_malloc (apoly.np*sizeof (MAV_vector));
  apoly.vert[0]= mav_vectorScalar (vecs[a], r);
  apoly.vert[1]= mav_vectorScalar (vecs[b], r);
  apoly.vert[2]= mav_vectorScalar (vecs[c], r);
  apoly.vert[3]= mav_vectorScalar (vecs[d], r);
  apoly.vert[4]= mav_vectorScalar (vecs[e], r);
  apoly.matrix= MAV_ID_MATRIX;

  /* Calculate line-polygon intersection */
  mav_linePolygonIntersection (&apoly, ln, oi);

  /* Free up polygon vertex memory */
  mav_free (apoly.vert);
}

/* Function to calculate the object-line intersection of the dodecahedron */
int mav_dodecIntersect (MAV_object *o, MAV_line *ln, MAV_objectIntersection *oi)
{
  MAV_dodec *dodec;
  MAV_objectIntersection pentInt[12];
  MAV_line ln2;

  /* Convert from generic Maverik object to the dodecahedron object */
  dodec= (MAV_dodec *) mav_objectDataGet (o);

  /* Initialise object intersection data structure */
  oi->pt1=-100.0;
```

```
    /* Rotate and translate line so that the dodecahedron
       is centered and axis-aligned */
    ln2= mav_lineTransFrame (*ln, dodec->matrix);

    /* Intersect the 12 pentagons that make up the dodecahedron */
    pentagonIntersect (ln2, &pentInt[0], 0, 1, 9, 16, 5, dodec->r);
    pentagonIntersect (ln2, &pentInt[1], 1, 0, 3, 18, 7, dodec->r);
    pentagonIntersect (ln2, &pentInt[2], 1, 7, 11, 10, 9, dodec->r);
    pentagonIntersect (ln2, &pentInt[3], 11, 7, 18, 19, 6, dodec->r);
    pentagonIntersect (ln2, &pentInt[4], 8, 17, 16, 9, 10, dodec->r);
    pentagonIntersect (ln2, &pentInt[5], 2, 14, 15, 6, 19, dodec->r);
    pentagonIntersect (ln2, &pentInt[6], 2, 13, 12, 4, 14, dodec->r);
    pentagonIntersect (ln2, &pentInt[7], 2, 19, 18, 3, 13, dodec->r);
    pentagonIntersect (ln2, &pentInt[8], 3, 0, 5, 12, 13, dodec->r);
    pentagonIntersect (ln2, &pentInt[9], 6, 15, 8, 10, 11, dodec->r);
    pentagonIntersect (ln2, &pentInt[10], 4, 17, 8, 15, 14, dodec->r);
    pentagonIntersect (ln2, &pentInt[11], 4, 12, 5, 16, 17, dodec->r);

    /* Sort intersection and return appropriate value */
    return (mav_objectIntersectionsSort (12, pentInt,
                                         mav_matrixScaleGet (dodec->matrix), oi));
}
```

As with the draw and BB methods, the intersect method is registered and executed respectively using `mav_callbackIntersectSet` (MFS p 260) and `mav_callbackIntersectExec` (MFS p 259).

We'll now look at the implementation of the dodecahedron intersection function. The first action taken is to initialise the `pt1` field of the object intersection data structure, `oi`, to some negative value. The next step is to transform the line, which is defined in the world coordinate frame, into the local coordinate frame of the dodecahedron. The mathematics of intersecting almost any object with a line is far easier if you can consider that object in its local coordinate frame, that is, centered at the origin and axis-aligned. This is achieved with the function `mav_lineTransFrame` (MFS p 273) which returns a `MAV_line` in the local coordinate frame given the world coordinate frame line and the transformation matrix between the local and world frames.

Like the rendering function, the intersection function considers the dodecahedron as 12 separate pentagons. It calculates the intersection with each of these and then sorts these to discover the closest point of intersection if any. Since this type of process is relatively common, MAVERIK provides a function, `mav_objectIntersectionsSort` (MFS p 277), which performs this sort. Its prototype is:

```
int mav_objectIntersectionsSort (int nhits, MAV_objectIntersection *hits,
                                 float scale, MAV_objectIntersection *res);
```

where `nhits` is the number of possible intersections, `hits` is the array of intersections, `scale` we will address in a moment and `res` is where to place the closest, if any, intersection. The return value of this function is `MAV_TRUE` if an intersection exists, or `MAV_FALSE` otherwise.

Since we are only interested in the distance to intersection, transforming the line between the two

coordinate frames makes no difference to the result, providing that the transformation only involves rigid body transformations such as translation and rotation.

If the transformation involves a scaling operation, then the distance to intersection has to be scaled appropriately. This is the purpose of the `scale` field in the `mav_objectIntersectionsSort` function. It is simply multiplied to the `pt1` value of the closest intersection to give the final value.

The scaling factor of a transformation matrix is returned by the function `mav_matrixScaleGet` (MFS p 274). Since MAVERIK only allows uniform scaling about the three axes, the single return value is sufficient to define this.

The intersection of a line with a pentagon is performed by creating a `MAV_polygon` (MFS p 23) to represent it. The MAVERIK polygon is fully described in Appendix B and comprises a number of points `np`, a normal `norm`, and an array containing the vertices, `vert`. Once in this form, we can calculate the line-polygon intersection with the function `mav_linePolygonIntersection` (MFS p 272). Note the use of `mav_malloc` (MFS p 149) and `mav_free` (MFS p 149) to allocate and release an area of memory. These are little more than wrappers to the standard `malloc` and `free` system calls, but the MAVERIK versions automatically check for `malloc` failing (stopping execution if it does) and keeps track of the amount of memory allocated and released to help with debugging memory leaks.

On execution, this example will appear to be very similar to the previous one. However, with careful positioning of the mouse is should be apparent that the keyboard event callback is only be executed when the mouse is truly, and not approximately, over the dodecahedron. Note that the `printf` has been removed from the rendering callback.

7.7 Example 13: other object callbacks

So far we have defined draw, calculate BB and calculate object-line intersection callbacks. The first two of these can viewed as the sensible minimum callbacks which need to be provided for a new object class. The third, while recommended, improves selection accuracy but does not add any intrinsically new functionality.

There are 6 other callbacks an object can define (by “object” we mean “a specified instance of the object class”):

- **delete**: called when an object is deleted giving you the opportunity to free any memory it used for example. Callback function prototype, `MAV_callbackDeleteFn` (MFS p 50):

  ```
  typedef int (*MAV_callbackDeleteFn) (MAV_object *);
  ```

- **identify**: return an identifier string for the object. Callback function prototype, `MAV_callbackIDFn` (MFS p 58):

  ```
  typedef int (*MAV_callbackIDFn) (MAV_object *, char **);
  ```

- **dump**: print a summary of the object's data structure to `stdout`. Callback function prototype, `MAV_callbackDumpFn` (MFS p 52):

```
typedef int (*MAV_callbackDumpFn) (MAV_object *);
```

- **getUserdef**: return a pointer to the user-defined data field of the object. Callback function prototype, `MAV_callbackGetUserdefFn` (MFS p 57):

```
typedef int (*MAV_callbackGetUserdefFn) (MAV_object *, void ***);
```

- **getMatrix**: return a pointer to the transformation matrix field of the object. Callback function prototype, `MAV_callbackGetMatrixFn` (MFS p 55):

```
typedef int (*MAV_callbackGetMatrixFn) (MAV_object *, MAV_matrix **);
```

- **getSurfaceParams**: return a pointer to the surface parameters field of the object. Callback function prototype, `MAV_callbackGetSurfaceParamsFn` (MFS p 56):

```
typedef int (*MAV_callbackGetSurfaceParamsFn) (MAV_object *, MAV_surfaceParams ***);
```

With the exception of delete, these callbacks are never executed by MAVERIK but rather by the application itself. The identify and dump callbacks are basically for debugging purposes while the "get" family of callbacks are used to obtain the data fields common to most objects. An example the "get matrix" and "get surface parameters" callbacks was shown in `eg7` to make the various objects "jump" and change colour.

Example 13 extends Example 7 (page 40) to include the dodecahedron in the scene. The "get matrix" and "get surface parameters" are implemented as follows so that the dodecahedron responds to the 'j', 'c' and 'p' keys to make it jump, change colour and be positioned by the mouse.

```
/* Function to return a pointer to the matrix field of the dodecahedron */
int mav_dodecGetMatrix(MAV_object *o, MAV_matrix **m)
{
  MAV_dodec *dodec= (MAV_dodec *) mav_objectDataGet (o);

  *m= &dodec->matrix;

  return MAV_TRUE;
}

/* Function to return a pointer to the surfaceParams field of the dodecahedron */
int mav_dodecGetSurfaceParams (MAV_object *o, MAV_surfaceParams ***sp)
{
  MAV_dodec *dodec= (MAV_dodec *) mav_objectDataGet (o);

  *sp= &dodec->sp;

  return MAV_TRUE;
}
```

Note that the rendering function for a pentagon has been modified to give a texture coordinate to each vertex. MAVERIK's texture coordinate data type, `MAV_texCoord` (MFS p 89), comprises two floats, `s` and `t`. They are applied using the function `mav_gfxTexCoord` (MFS p 268) which is analogous to the OpenGL function `glTexCoord2f`. Using a set of textured surface parameters on an object which does not define texture coordinates leads to undefined results.

The texture coordinates of a pentagon are arbitrarily chosen to be (sin(`ang`), cos(`ang`)) where `ang` starts at 0 and increments by 72 degrees each vertex $(72 = 360/5)$.

7.8 Example 14: redefining object callbacks

Of course there is nothing stopping you redefining the callbacks of the default MAVERIK objects to be your own functions. Example 14 demonstrates this by taking Example 5 (page 35) and redefining the draw callback for the box to be:

```
/* New box draw callback */
int myBoxDraw (MAV_object *o, MAV_drawInfo *di)
{
  /* Print a message to the shell window */
  printf ("In new box draw callback\n");

  /* Call the original draw callback function to render the box */
  mav_boxDraw (o, di);

  return MAV_TRUE;
}
```

and this is set in the main function with:

```
/* Redefine draw callback for boxes */
mav_callbackDrawSet (mav_win_all, mav_class_box, myBoxDraw);
```

Obviously, we could have chosen to actually render the object in the new draw callback maybe using triangles instead of quads for the surfaces since this may be quicker on some particular hardware configuration.

The names of the functions which act as the callbacks to the MAVERIK objects are given in Section B. A novel feature of MAVERIK is that the source code to these is available to the application programmer to inspect, copy and modify. The source code to the 19 MAVERIK objects can be found in the `src/objects` sub-directory of the MAVERIK distribution. As mentioned previously, the MAVERIK objects should be seen as "defaults": hopefully useful as is and a good starting point for customization when they don't fit the application's exact requirements.

7.9 Example 15: using “drawing information”

Example 15 modifies Example 10 (page 72) to use the drawing information in the draw callback to apply level of detail (LOD) in rendering the dodecahedron.

The MAVERIK drawing information data structure, `MAV_drawInfo`, contains three fields: the view frustum clip planes, `cp` (of data type `MAV_clipPlanes` (MFS p 68)), the view parameters, `vp`, and a user definable field, `userdef`.

The first of these fields could be used to determine which parts of an individual object are visible when the object’s BB is only partially inside the view frustum. This, and the third field, would only be used by advanced users.

It is the second field, the view parameters, which we shall use to apply LOD based on the distance of the object from the eye position. Unfortunately, unlike a cylinder where you can change the faceting accuracy, there is not a lot you can do to simplify a dodecahedron. Our approach, which is rather contrived but shows the principles, is to define a new dodecahedron rendering callback as follows:

```
/* Function to render the dodecahedron with level of detail */
int mav_dodecDrawLOD (MAV_object *o, MAV_drawInfo *di)
{
  MAV_dodec *dodec;
  float dist;

  /* Convert from generic Maverik object to the dodecahedron object */
  dodec=  (MAV_dodec *) mav_objectDataGet (o);

  /* Calculate distance from eyepoint */
  dist= sqrt (mav_vectorDotProduct (di->vp.eye, di->vp.eye));

  if (dist<50)
  {
    /* Full detail */
    mav_dodecDraw (o, di);

    printf ("Full detail\n");
  }
  else if (dist<100)
  {
    /* Draw as sphere  */
    MAV_sphere s;
    MAV_object so;

    s.radius= dodec->r;
    s.nverts= 4;
    s.nchips= 4;
    s.sp= dodec->sp;
    s.matrix= dodec->matrix;

    so.the_class= mav_class_sphere;
```

```
      so.the_data= &s;

      mav_sphereDraw (&so, di);

      printf ("Sphere\n");
    }
    else if (dist<150)
    {
      /* Wire frame draw */
      mav_windowPolygonModeSet (mav_win_all, MAV_POLYGON_LINE);
      mav_dodecDraw (o, di);
      mav_windowPolygonModeSet (mav_win_all, MAV_POLYGON_FILL);

      printf ("Wire frame\n");
    }
    else
    {
      printf ("Not drawing\n");
    }

    return MAV_TRUE;
  }
```

The distance from the eye point to the center of the dodecahedron is calculated. This calculation is only valid since the dodecahedron is centered at the origin. If it were not, then the draw information could be translated into the local coordinate frame using the function `mav_drawInfoTransFrame` (MFS p 265):

```
MAV_drawInfo mav_drawInfoTransFrame (MAV_drawInfo in, MAV_matrix mat);
```

This is completely analogous to `mav_lineTransFrame` and translates a `MAV_drawInfo` data type between coordinate frames.

The LOD metric we apply is to render the dodecahedron in full detail if it is less than 50 units away; as a sphere if the distance is greater than 50 but less than 100; a wireframe dodecahedron if the distance is greater than 100 but less than 150; and not to draw it at all it if the distance is greater than 150. These values are chosen to show, not hide, the changes. (And yes, we know, that the sphere is probably more costly to render than the dodecahedron in full detail – but that isn't the point!)

Note how a sphere is created to represent the dodecahedron. The function to render a sphere must be called with a `MAV_object` rather than directly with a `MAV_sphere` (MFS p 28). We could of course register the sphere as a MAVERIK object, render it, and then delete it, but this is wasteful. A better solution is to implicitly create a temporary MAVERIK object filling in the required fields of this data structure ourselves. `MAV_objects` consist of two fields: a `MAV_class` pointer, `the_class`, to define the methods which operate on the data portion, a void pointer `the_data` (the "the_" is included to avoid a clash with the `class` reserved keyword in C++). This type of MAVERIK object creation should be used sparingly and strictly limited to the type of operation performed here, that is, where a temporary object is required and used to perform a specific known task. You should never use MAVERIK objects created in this manner to execute callbacks or add them to an SMS.

The wireframe dodecahedron is drawn by toggling the state of the polygon mode using the function `mav_windowPolygonModeSet` (MFS p 228). (This is the same function called when pressing Shift-F8 to toggle between wireframe and filled mode).

Chapter 8

Customising navigation

In this chapter we illustrate how to write new navigator functions and detail exactly how navigation events are detected and handled.

8.1 Navigator functions

Recall from Example 8 (page 48) that customization of the default mouse navigation behaviour is via the function:

```
void mav_navigationMouseDefaultParams(MAV_window *w, int but,
                                      MAV_navigatorFn x, float xls, float xas,
                                      MAV_navigatorFn y, float yls, float yas);
```

A `MAV_navigatorFn` (MFS p 82) has the following prototype:

```
typedef void (*MAV_navigatorFn)(MAV_viewParams *vp, float am, float ls, float as);
```

It modifies the contents of the view parameters, `vp`, by an amount, `am`. `am` is scaled by `ls` to convert it in to application units in order to apply linear transformations; and by `as` to convert it in to radians in order to apply rotational transformations.

The controlling code for the default mouse navigation, which we will describe in Section 8.4, executes the appropriate horizontal and vertical navigator functions with an `am` equal to the distance in pixels that the mouse has travelled in that direction.

It is easiest to illustrate navigator functions by looking at how some of the default navigator functions are implemented. The source code for these can be found in the `mav_navigators.c` in the `src/navigation` sub-directory of the MAVERIK distribution.

`mav_navigateTransX` (MFS p 164) simply modifies the X coordinate of the eye point:

```
void mav_navigateTransX(MAV_viewParams *vp, float amount, float ls, float as)
{
  /* x axis shift */
  vp->eye.x += (amount*ls);
}
```

The two related navigator functions, `mav_navigateTransY` (MFS p 164) and `mav_navigateTransZ` (MFS p 164), are similarly defined.

`mav_navigateForwards` (MFS p 164) applies each of `mav_navigateTransX`, `mav_navigateTransY` and `mav_navigateTransZ` to move the eye point along the view direction vector:

```
void mav_navigateForwards(MAV_viewParams *vp, float amount, float ls, float as)
{
  /* view direction shift */
  mav_navigateTransX(vp, vp->view.x * amount, ls, as);
  mav_navigateTransY(vp, vp->view.y * amount, ls, as);
  mav_navigateTransZ(vp, vp->view.z * amount, ls, as);
}
```

`mav_navigateYaw` (MFS p 164) illustrates how the view direction can be modified:

```
void mav_navigateYaw(MAV_viewParams *vp, float amount, float ls, float as)
{
  /* yaw */
  vp->view= mav_vectorRotate(vp->view, vp->up, amount*as);
  vp->right= mav_vectorRotate(vp->right, vp->up, amount*as);
}
```

`mav_vectorRotate` (MFS p 210) rotates the first parameter about the second by an amount given by the third parameter in radians.

8.2 Example 16: simple collision detection

Example 16 (`eg16.c`) modifies Example 13 (page 85) to include the following navigator function which performs simple collision detection:

```
/* Navigator function with collision detection */
void myNavigator(MAV_viewParams *vp, float am, float ls, float as)
{
  MAV_viewParams orig;
  MAV_line ln;
  MAV_object *o;
  MAV_objectIntersection oi;
```

```
    float dist;

    /* Copy the original view parameters */
    orig= *vp;

    /* Navigate forwards */
    mav_navigateForwards(vp, am, ls, as);

    /* Calculate the direction and distance of travel */
    ln.pt= orig.eye;
    ln.dir= mav_vectorSub(vp->eye, orig.eye);
    dist= sqrt(mav_vectorDotProduct(ln.dir, ln.dir));
    ln.dir= mav_vectorNormalize(ln.dir);

    /* Check if any objects intersect this line */
    if (mav_SMSIntersectLineAll(mav_win_current, ln, &oi, &o)) {
      /* Is the intersection closer than the distance travelled? */
      if (oi.pt1 < (dist+3.0)) {
        /* Collision occurred, so use original view parameters */
        *vp= orig;
        printf("Collision occurred\n");
      }
    }
  }
```

The default mouse navigation is made to use this navigator function by calling:

```
  /* Use customized navigation */
  mav_navigationMouseDefaultParams(mav_win_all, MAV_LEFT_BUTTON,
                                   mav_navigateYawFixedUp, 0.01, -0.0005,
                                   myNavigator, 0.01, 0.0005);
```

The collision detection works by intersecting all SMS's (in other words, all objects) with the line which joins the eyepoint before and after the navigation has been applied. If any object intersects this line, and the distance to the intersection point is less than the distance travelled, then a collision has occurred and the navigator does not modify the view parameters. Note that an arbitrary constant is used to prevent the eyepoint from getting too close to an object which would otherwise fill the field of view and disorientate the user.

This is a very simplistic implementation of collision detection. Collision only occurs if the movement in the eyepoint intersects an object. A more realistic test would be to check that a particular volume of space, representing the users body, does not intersect an object.

Furthermore, collision detection is only performed on one navigator function, and it would be impractical to implement collisions detection in this manner for all the others.

We will address these two issues in the remainder of this chapter.

8.3 Events

In this section we describe how mouse events trigger navigation.

MAVERIK has two separate mouse event callback functions. The one described so far, introduced in Example 5 (page 35), is used by the application to define an object's response to mouse events. The other is reserved specifically for implementing mouse navigation. It is defined by the second argument of `mav_navigationMouse` and is triggered when any mouse button events occurs anywhere in the specified window.

When a mouse button event occurs, the function set with `mav_navigationMouse` is executed. If no such function was set or its return value was `MAV_FALSE`, then the mouse event callback function set with `mav_callbackMouseSet` is also executed if its applicable.

A mouse event callback to implement navigation could be defined as follows:

```
int myXOrig, myYOrig;

void myMove(void *ignored)
{
  float xdiff, ydiff;

  /* Calculate amount mouse has moved from navigation origin */
  xdiff= mav_mouse_x-myXOrig;
  ydiff= -(mav_mouse_y-myYOrig);

  /* Apply navigator functions */
  mav_navigateYaw(mav_win_current->vp, xdiff, 0.01, -0.0005);
  mav_navigateForwards(mav_win_current->vp, ydiff, 0.01, 0.0005);
}

int myNav(MAV_object *o, MAV_mouseEvent *me)
{
  if (me->movement==MAV_PRESSED)
  {
     /* Note origin of navigation */
     myXOrig= me->x;
     myYOrig= me->y;

     /* Start executing myMove function at beginning of frame */
     mav_frameFn0Add(myMoveFn, NULL);
  }
  else
  {
     /* Stop executing myMove function */
     mav_frameFn0Rmv(myMove, NULL);
  }

  /* Also pass event onto application defined mouse event callback fn */
  return MAV_FALSE;
```

```
}
```

The mouse event callback is set using:

```
mav_navigationMouse(mav_win_all, myNav);
```

The principle here is that when a mouse event occurs, the mouse position is noted and a function is set to be executed at the beginning of every frame. This function calculates the horizontal and vertical displacements between the current mouse position and its noted position. These then act as the amounts by which the navigator functions modify the view parameters. Note how the vertical mouse displacement is calculated due to windows having their top-left corner as the origin.

Obviously, the collision detection check could now be incorporated in to `myMove` and hence would work with any navigator function you chose to place in this function.

8.4 Default mouse navigation

The MAVERIK default mouse navigation function, `mav_mouseNavigationDefault`, is a bit more complicated than the one shown above. As well as enabling you to define the navigator functions and scaling factors, it also allows for more than one navigator function to be active at once. (The implementation shown above would fail badly if a second button was pressed before the first was released).

The source code to `mav_mouseNavigationDefault` can be found in `mav_mouse.c` in the `src/navigation` sub-directory of the MAVERIK distribution.

An alternative form of navigation could be achieved by always having the function `myMove` executed at the start of the frame and setting the navigation origin to the middle of the screen. This would give constantly active navigation, which is not triggered by mouse button events.

Keyboard navigation is implemented in exactly the same manner.

8.5 Example 17: complex collision detection

Example 17 (`eg17.c`) modifies Example 16 to implement the type of navigation event handler described above.

```
int myXOrig[3], myYOrig[3];

void myMove(int i)
{
  MAV_viewParams orig;
```

```
  MAV_BB bb;
  float xdiff, ydiff;
  MAV_SMS *tmp;

  /* Copy the original view parameters */
  orig= *(mav_win_current->vp);

  /* Calculate amount mouse has moved from navigation origin */
  xdiff= mav_mouse_x-myXOrig[i];
  ydiff= -(mav_mouse_y-myYOrig[i]);

  /* Apply navigator functions */
  switch (i) {
    case 0: /* left button */
      mav_navigateYaw(mav_win_current->vp, xdiff, 0.01, -0.0005);
      mav_navigateForwards(mav_win_current->vp, ydiff, 0.01, 0.0005);
      break;
    case 2: /* right button */
      mav_navigateRight(mav_win_current->vp, xdiff, 0.01, -0.0005);
      mav_navigateUp(mav_win_current->vp, ydiff, 0.01, 0.0005);
      break;
  }

  /* Make a BB of size 2 units around the eye point */
  bb.min=mav_vectorAdd(mav_win_current->vp->eye, mav_vectorSet(-1.0, -1.0, -1.0));
  bb.max=mav_vectorAdd(mav_win_current->vp->eye, mav_vectorSet(+1.0, +1.0, +1.0));

  /* Create a temporary objList SMS to hold any objects which intersects this BB */
  tmp= mav_SMSObjListNew();

  /* Check if this BB intersects any object */
  if (mav_SMSIntersectBBAll(mav_win_current, bb, tmp)) {
    /* Collision occurred, so use original view parameters */
    *(mav_win_current->vp)= orig;
    printf("Collision occurred\n");
  }

  /* Delete temporary SMS (but not its contents) */
  mav_SMSDelete(tmp, MAV_FALSE);
}

void myMove0(void *ignored) {
  myMove(0);
}

void myMove1(void *ignored) {
  myMove(1);
}

void myMove2(void *ignored) {
  myMove(2);
}
```

```
MAV_frameFn myMoveFn[]={myMove0, myMove1, myMove2};

int myNav(MAV_object *o, MAV_mouseEvent *me)
{
  if (me->movement==MAV_PRESSED)
  {
     /* Note origin of navigation */
     myXOrig[me->button]= me->x;
     myYOrig[me->button]= me->y;

     /* Start executing myMove function at beginning of frame */
     mav_frameFn0Add(myMoveFn[me->button], NULL);
  }
  else
  {
     /* Stop executing myMove function */
     mav_frameFn0Rmv(myMoveFn[me->button], NULL);
  }

  /* Also pass event onto application defined mouse event callback fn */
  return MAV_FALSE;
}
```

This is set by:

```
/* Use customized mouse navigation */
mav_navigationMouse(mav_win_all, myNav);
```

Note the use of arrays to allow more than one button to be pressed simultaneously.

The collision detection in this example uses an axis-aligned bounding box of size 2 units centered at the position the user is trying to move to. The function `mav_SMSIntersectBBAll` (MFS p 237) determines if that BB intersects any object and if so adds the them to an SMS. (In this example we are not concerned with which object(s) the BB encompasses, only that it does, and so the SMS is instantly deleted with `mav_SMSDelete` (MFS p 126)). This gives rise to a more realistic collision.

Chapter 9

Working with an SMS

In would be nice to take a bird's eye view of Example 17 (page 95) and render the BBs of each of the objects, so we could see the collision detection process happening.

We already know how to calculate and display the BB for a single object – see Example 11 (page 79); what we need is a method for applying this operation (or indeed any other) to the entire contents of an SMS.

This chapter describes how this can be achieved.

9.1 The implementation of an SMS

An SMS is implemented in exactly the same way as an object class: using callback functions. Both an SMS and an object are simply data structures with associated functions which perform certain operations on the data.

As with object classes, applications are free to create their own SMS data structures and define the callback functions which operate on them.

But for now, all we need to describe is a subset of the SMS callbacks and how they can be executed. A fuller description of SMS's is presented in Chapter 13.

9.2 Basic SMS callbacks

You have already, albeit unknowingly, encountered two callbacks which operate on an SMS:

- The "add object" callback, executed by the function `mav_SMSCallbackObjectAddExec` (MFS p 292), which has the prototype:

```
int mav_SMSCallbackObjectAddExec(MAV_SMS *s, MAV_object *o);
```

and adds object `o` to SMS `s`. The function `mav_SMSObjectAdd` is simply a wrapper to the above call and hides the novice user from an otherwise confusing function name.

- The "remove object" callback, executed by the function `mav_SMSCallbackObjectRmvExec` (MFS p 294), which has the prototype:

```
int mav_SMSCallbackObjectRmvExec(MAV_SMS *s, MAV_object *o);
```

and removes object `o` from SMS `s`. The function is executed when an object in deleted with `mav_objectDelete` to remove it from any SMS's it is in. The function `mav_SMSObjectRmv` is simply a wrapper to the above call.

9.3 The "reset" and "next" SMS callbacks

Regardless of the kind of data structure comprising the SMS – for example, a simple linked list, or a complex hierarchical structure – it can be traversed one element at a time.

Two callbacks are defined, executed with `mav_SMSCallbackPointerResetExec` (MFS p 295) and `mav_SMSCallbackObjectNextExec` (MFS p 295), to allow an application to do this:

```
int mav_SMSCallbackPointerResetExec(MAV_SMS *s);

int mav_SMSCallbackObjectNextExec(MAV_SMS *s, MAV_object **o);
```

Executing the first function resets an internal pointer kept by the SMS. The second function obtains the object at the internal pointer's location and increments it to the next object in the SMS's structure. The function returns `MAV_FALSE` when the SMS has been fully traversed.

So, an SMS can be traversed as follows:

```
MAV_SMS *sms;
MAV_object *obj;

mav_SMSCallbackPointerResetExec(sms);
while (mav_SMSCallbackObjectNextExec(sms, &obj)) {
  /* perform operation on object 'obj' */
}
```

Warning: in general you should not add objects to, or remove objects from, an SMS while it is being traversed, since these operations may invalidate the internal pointer. If for example you need to remove certain objects from an SMS, first traverse the SMS and note all the relevant objects in a separate list. Then, when the SMS has been fully traversed, traverse the separate object list and then remove the objects it contains from the SMS. MAVERIK lists provide a convenient means of achieving this – see `mav_listNew` (MFS p 147).

9.4 The push and pop SMS callbacks

It is sometime necessary to recursively traverse an SMS – to pass through it one (or more) times for each object in the SMS. Such a pass would be needed, say, to calculate the closest neighbour to each object.

Two callbacks are defined to achieve this, executed by `mav_SMSCallbackPointerPushExec` (MFS p 295) and `mav_SMSCallbackPointerPopExec` (MFS p 295) which have the prototypes:

```
int mav_SMSCallbackPointerPushExec(MAV_SMS *s);

int mav_SMSCallbackPointerPopExec(MAV_SMS *s);
```

These functions respectively push and pop the internal pointer using a stack maintained by the SMS.

For example:

```
MAV_SMS *sms;
MAV_object *obj1, *obj2;

mav_SMSCallbackPointerResetExec(sms);
while (mav_SMSCallbackObjectNextExec(sms, &obj1)) {

  mav_SMSCallbackPointerPushExec(sms);
  mav_SMSCallbackPointerResetExec(sms);
  while (mav_SMSCallbackObjectNextExec(sms, &obj2)) {

    /* operations on objects 'obj1' and 'obj2' */

  }
  mav_SMSCallbackPointerPopExec(sms);
}
```

Currently these operations only work on the "object list" type of SMS. They have not yet been implemented for the hierarchical bounding box SMS.

9.5 The "execute function" SMS callback

We could use the callbacks described above to traverse the SMS in Example 17 (page 95) to calculate and render the BBs of its contents. But this would be wasteful since it would consider all objects in the SMS, not just those which are visible in the view frustum.

You may be surprised to learn that there is no "cull objects and display" SMS callback – we felt this was far too specific. What there is however is an "execute function" SMS callback. This culls

the objects in the SMS to a set of arbitrary clip planes, executing a further function on the resulting objects. It's a callback within a callback!

A wrapper function can than made which uses this callback with the clip planes that correspond to the view frustum and arranges so that the function executed displays the object. But, the same callback can be used – for example – to determine which objects are within a region of space around a user's avatar and have these objects change colour.

The execution of the callback which achives this is via the command `mav_SMSCallbackExecFnExec` (MFS p 288) and has the prototype:

```
int mav_SMSCallbackExecFnExec(MAV_SMS *s, MAV_drawInfo *di, MAV_SMSExecFn *fn);
```

The two data structure arguments to this function are as follows:

```
typedef struct {
  MAV_clipPlanes cp;
  MAV_viewParams vp;
  void *userdef;
} MAV_drawInfo;

typedef void (*MAV_SMSExecFnFn)(MAV_object *, MAV_drawInfo *, void *);

typedef struct {
  MAV_SMSExecFnFn fn;
  int nocalc;
  void *params;
} MAV_SMSExecFn;
```

The `MAV_drawInfo` data structure contains the culling information which comprises:

- A set of clip planes. Creating your own clip planes can be tricky, but MAVERIK provides a function to create these from the view frustum (see below).
- The current view parameters. An SMS may also perform some LOD processing, e.g. only passing on objects which are within the clip planes and are closer that some threshold to the eye.
- A user-definable field which an application can use to hold any extra data relevant to the culling operation. For example, if you have an eye-tracking system it could contain which object the user is looking at.

The `MAV_SMSExecFn` (MFS p 43) data structure defines what function is to be executed on the objects which pass the cull test. This comprises:

- The function to execute (of type `MAV_SMSExecFn` (MFS p 43) – prototype shown above).

- An indication of whether or not the callback function should be executed if the required calculation on the object can not be performed (for example its bounding box callback is not set).
- A user-definable field. This is not interpreted by the SMS but rather forms the third field of the function to execute. This field could be used, for example, to hold a MAVERIK list which the object is then added to.

It may be easier to appreciate the workings of the "execute function" callback when seen in action. Below is a representation of how `mav_SMSDisplay` could be implemented:

```
void disp(MAV_object *o, MAV_drawInfo *di, void *ignored)
{
  /* Execute the draw callback of the object */
  mav_callbackDrawExec(o, di);
}

void mav_SMSDisplay(MAV_window *w, MAV_SMS *sms)
{
  MAV_SMSExecFn fn;
  MAV_drawInfo di;

  /* Make an SMSExecFn to call 'disp' on objects which pass the cull */
  fn.fn= disp;
  fn.nocalc= 1; /* If unsure call 'disp' anyway */
  fn.params= NULL;

  /* Make up draw info which corresponds to the view */
  di.cp= mav_clipPlanesGet(w, -1.0, 1.0, -1.0, 1.0, w->ncp / w->fcp, 1.0);
  di.vp= *(w->vp);

  mav_SMSCallbackExecFnExec(sms, &di, &fn);
}
```

`mav_clipPlanesGet` (MFS p 264) creates a set of clip planes from the view frustum in a window. The various arguments to this function allow for clip planes to be defined using only a portion of the view frustum.

The actual implementation of `mav_SMSDisplay` is such that it does not execute a fixed function; rather it executes the function defined by the global variable `mav_SMS_displayFn`. By default, this variable is set to be the function `mav_SMSDisplayFn` (MFS p 235) which simply executes the draw callback of the object.

9.6 Example 18: collision detection revisited

Example 18 (`ex18.c`) modifies Example 17 (page 95) to render the BBs of each of the objects so that we can see the collision detection process happening.

It achieves this by setting the `mav_SMS_displayFn` variable to point to its own function which renders the object and calculates and displays their BB.

```
/* Customized SMS display function */
void myDraw(MAV_object *o, MAV_drawInfo *di, void *ignored)
{
  MAV_BB bb;

  /* Draw object */
  mav_callbackDrawExec(mav_win_current, o, di);

  /* Calculate and display BB */
  mav_callbackBBExec(mav_win_current, o, &bb);
  mav_BBDisplay(mav_win_current, bb);
}
```

which is set in `main` by:

```
/* Set the SMS display function */
mav_SMS_displayFn= myDraw;
```

This example opens an additional window with its own independent set of viewing parameters:

```
MAV_window *ov;
MAV_viewParams ovp;

/* Create an overview window with its own view params */
ov= mav_windowNew(200, 200, 200, 200, "overview", NULL);
mav_windowViewParamsSet(ov, &ovp);
```

The view in this second window is defined to be 50 units above the view point of the first window and looking straight down to give an overview effect. The BB used for collision detection is rendered in the overview window.

Both of these features are achieved using frame modification functions:

```
void fixView(void *ignored)
{
  /* Copy the other window's view */
  ovp= mav_vp_default;

  /* But move vertical up and look down */
  ovp.eye.y+=50;
  ovp.view= mav_vectorSet(0,-1,0);
  ovp.up= mav_vp_default.view;
}
```

```
void drawBB(void *ignored)
{
  /* Display user’s BB in overview window */
  mav_BBDisplay(ov, bb);
}
```

These are set in `main` with:

```
/* Functions to set the overview window view and render the BB */
mav_frameFn1Add(fixView, NULL);
mav_frameFn2Add(drawBB, NULL);
```

9.7 The “all class” handle

Another way of achieving the effect of Example 17 would be to exploit the properties of the object class `mav_class_all`. Callbacks set for this class take priority over ones set for a specific class. For example, we could define a draw callback as follows:

```
int myDraw(MAV_object *o, MAV_drawInfo *di)
{
  MAV_BB bb;

  /* Clear the "all classes" draw callback */
  mav_callbackDrawSet(mav_win_all, mav_class_all, NULL);

  /* Draw object */
  mav_callbackDrawExec(mav_win_current, o, di);

  /* Calculate and display BB */
  mav_callbackBBExec(mav_win_current, o, &bb);
  mav_BBDisplay(mav_win_current, bb);

  /* Reset the "all classes" draw callback */
  mav_callbackDrawSet(mav_win_all, mav_class_all, myDraw);
}
```

This would be set in `main` with:

```
/* Set the "all classes" draw callback */
mav_callbackDrawSet(mav_win_all, mav_class_all, myDraw);
```

Note the need to clear the “all classes” draw callback to prevent an infinite loop occurring.

Chapter 10

Defining new object callbacks

Chapter 7 described the standard set of callback methods which MAVERIK defines, which include draw object, calculate BB of object, and so on. While these are general methods needed to process objects in a virtual environment, some applications may wish to define their own methods, specific to their needs. For example, a CAD application might wish to:

- define a method to obtain the part number of an object, which could then be displayed alongside the object;
- define a method to obtain particular attributes of an object. For example, when modelling pipes, taking different actions depending on whether a pipe is known to be full of liquid or gas. This might be used to only allow an operator to interactively modify sections of pipe-work which are known to be empty;
- define a method which is called during the manipulating of an object so as to enforce some constraint, such as minimum inter-object separation, or health and safety regulations.

MAVERIK allows applications to extend its default set of callback methods to incorporate such application-specific methods. However, since an application would need to both define and execute the callback function itself – MAVERIK will make no use of it – it is a subtle point as to the advantage of processing objects via a callback mechanism, as opposed to simply calling the relevant function in the usual way. The advantage is the same as that described in Chapter 7 for creating application specific classes of object: encapsulation.

New callback methods can be encapsulated and effectively incorporated into the MAVERIK system so that it would *appear* to another user that MAVERIK is making use of this functionality, whereas in fact it is only being used by a layer on top of the MAVERIK kernel. By now you may have guessed that the "standard" MAVERIK callback methods, like the "default" objects, are not part of the MAVERIK kernel at all – but have been added on later using the mechanism we are about to describe.

It is easier to appreciate the encapsulation issue when considering event-based callbacks (recall that in MAVERIK event-based and process-based callbacks are implemented by the same mechanism). In

supporting a new input device, for example, it would be essential that a new event-based callback be created to allow objects to define their response to events generated by the device. This, along with the controlling code for the device which detects events and executes the callbacks, could then be encapsulated and given to other users to provide support for the input device. We'll discus adding MAVERIK support for new input devices in Chapter 12.

10.1 Example 19: the "calculate volume" callback

To demonstrate the process of creating a new callback method we shall implement a "calculate volume" callback. This is a new callback which object classes can define to calculate and return the volume of the object.

This could be used for example in Example 7 (page 40), so that only objects less than a certain volume would be allowed to "jump" or be "picked" with the "j" and "p" keys. Similarly, it could be used in the collision detection navigation (Chapter 8) so that small objects do not constitute a collision which stops the user's movements.

A new callback is obtained using the function `mav_callbackNew` (MFS p 305) which returns a pointer to a `MAV_callback` (MFS p 65) data structure which acts as the handle to the newly created callback:

```
MAV_callback *mav_callback_calcVol;

mav_callback_calcVol= mav_callbackNew();
```

Note that we are using a naming scheme consistent with the "standard" MAVERIK callbacks thus making any potential encapsulation more transparent.

The exact contents of the `MAV_callback` data structure are of little concern to the average user. (In fact this data structure simply contains a unique integer which acts as an index into an array of callback functions held for each class of object).

A callback function is set for a particular class of objects using `mav_callbackSet` (MFS p 307), as follows:

```
void mav_callbackSet(MAV_callback *cb, MAV_window *w, MAV_class *c, MAV_callbackFn fn);
```

where `cb` specifies the callback, `w` the window (callback functions are defined on a per-window basis), `c` is the class of object and `fn` the callback function.

Callback function, `MAV_callbackFn` (MFS p 54), are of type:

```
typedef int (*MAV_callbackFn)(MAV_object *o, void *d1, void *d2);
```

where `o` is the object to process and `d1` and `d2` are void pointers the meaning of which we'll describe shortly.

The function set for callback `cb`, in window `w`, for object `o` can then be executed using `mav_callbackExec` (MFS p 304):

```
int mav_callbackExec(MAV_callback *cb, MAV_window *w, MAV_object *o, void *d1, void *d2);
```

`d1` and `d2` are typically pointers to callback-specific data structures cast to be void pointers. These values are passed untouched to the callback function which re-casts them to be pointers to the appropriate data structures. This allows up to two pointers to arbitrary data structures to be passed to the callback functions – usually one to detail any additional information needed by the callback function and the other to detail the results of the calculation.

Take for example the object-line intersection callback: the information passed into this callback function is a `MAV_line` data structure which defines the line for intersection testing, and a `MAV_objectIntersection`, which details the results of the intersection calculation. While these two data structures could be combined into one, keeping the input and output data structures separate makes sense. Of course, `NULL` values can be used for a particular callback if an input and/or output data structure is not applicable.

The return value of `mav_callbackExec` is the return value of the callback function or `MAV_FALSE` if no callback function has been set.

10.2 Callback wrappers

In their "raw" form described above, callbacks are somewhat cumbersome to use. To overcome this, additional "wrapper" functions can be defined to set and execute the callback with tighter prototyping:

```
typedef int (*MAV_callbackCalcVolFn)(MAV_object *o, float *vol);

void mav_callbackCalcVolSet(MAV_window *w, MAV_class *c, MAV_callbackCalcVolFn fn)
{
  mav_callbackSet(mav_callback_calcVol, w, c, (MAV_callbackFn) fn);
}

int mav_callbackCalcVolExec(MAV_window *w, MAV_object *o, float *vol)
{
  return (mav_callbackExec(mav_callback_calcVol, w, o, vol, NULL));
}
```

The box class could then define a function to calculate the volume of this shape as follows:

```
int mav_boxCalcVol(MAV_object *o, float *vol)
```

```
{
  /* Convert from generic Maverik object to a box object */
  MAV_box *box= (MAV_box *) mav_objectDataGet(o);

  /* Calculate volume */
  *vol= (box->size.x*box->size.y*box->size.z);

  return MAV_TRUE;
}
```

which would be set with:

```
mav_callbackCalcVolSet(mav_win_all, mav_class_box, mav_boxCalcVol);
```

and executed as follows:

```
float vol;

mav_callbackCalcVolExec(mav_win_current, obj, &vol);
```

Example 19 (`eg19.c`) is a modified version of Example 5 (page 35), which implements the “calculate volume” callback printing the results of the calculation to the shell window.

10.3 Example 20: the “calculate volume” callback extended

Example 20 (`eg20.c`) modifies Example 17 (page 95) to implement the “calculate volume” callback for the box and cylinder.

The volume of a cylinder is trivially calculated as follows:

```
/* Routine to calculate the volume of a cylinder */
int mav_cylinderCalcVol(MAV_object *o, float *vol)
{
  /* Convert from generic Maverik object to a cylinder object */
  MAV_cylinder *cyl= (MAV_cylinder *) mav_objectDataGet(o);

  /* Calculate volume */
  *vol= (MAV_PI*cyl->radius*cyl->radius*cyl->height);

  return MAV_TRUE;
}
```

and set in the main routine with:

```
mav_callbackCalcVolSet(mav_win_all, mav_class_cylinder, mav_cylinderCalcVol);
```

This example uses the mav_SMS_displayFn variable (introduced in Example 18 (page 103) to point to its own function which renders the object and calculates and displays its volume.

```
/* New SMS display fn which calculates and displays the volume of an object */
void myDisp(MAV_object *o, MAV_drawInfo *di, void *ig)
{
  MAV_matrix *m;
  MAV_vector p;
  MAV_BB bb;
  char msg[100];
  float v;

  /* Draw object */
  mav_callbackDrawExec(mav_win_current, o, di);

  /* Calculate volume, either from callback or BB */
  if (mav_callbackQuery(mav_callback_calcVol, mav_win_current, o))
  {
    mav_callbackCalcVolExec(mav_win_current, o, &v);
  }
  else
  {
    mav_callbackBBExec(mav_win_current, o, &bb);
    v= (bb.max.x-bb.min.x)*(bb.max.y-bb.min.y)*(bb.max.z-bb.min.z);
  }

  /* Get position of object */
  mav_callbackGetMatrixExec(mav_win_current, o, &m);
  p= mav_matrixXYZGet(*m);

  /* Convert this to screen coordinates */
  p= mav_vectorScrnPos(p);

  /* Display the volume at this location (provided its in front of us) */
  if (p.z<1.0) {
    sprintf(msg, "vol= %.0f\n", v);
    mav_stringDisplay(mav_win_current, msg, MAV_COLOUR_BLACK, 0, p.x, p.y);
  }
}
```

This routine checks if the "calculate volume" callback has been defined for the object. If it has, its executed to obtain the volume of the object. If not, the BB is used to estimate the volume.

A text string containing the volume is printed to the screen. The function mav_vectorScrnPos (MFS p 212) returns the NDC screen coordinates of the supplied world coordinate frame vector.

This routine is set in main by:

```
/* Override default SMS display function */
mav_SMS_displayFn= myDisp;
```

Chapter 11

Miscellaneous Level 2 topics

We've now come to the end of the worked examples for the MAVERIK Programming Level 2 section of this manual. This chapter describes various miscellaneous concepts and function calls that a Level 2 programmer may need to know about, but which have not been covered by the worked examples.

This chapter should be considered as an appendix or reference section to Programming Level 2. Some repetition of the material in the earlier chapters is inevitable.

11.1 The hierarchical bounding box SMS

The MAVERIK distribution provides two types of SMS. So far we have only discussed the "Object List" SMS, which stores objects in the order in which they were inserted, and uses the bounding boxes of those objects to determine if they are visible and thus should be rendered. This section describes the other type of SMS, the hierarchical bounding box (HBB) SMS. Chapter 13 describes how new types of SMS can be defined which store and process objects in an application-specific manner.

HBBs are designed to store the static objects in a scene – objects whose sizes and positions don't change. This restriction means that upon insertion the object's axis-aligned bounding box (BB) can be calculated and stored for efficiency rather than being re-calculated each time it is required, which is the case for the object list SMS.

Furthermore, as objects are inserted, a hierarchy of BBs is built up with each object's BB as a leaf node. Testing a BB determines the action required for all the objects beneath it in the hierarchy. In the case of view-frustum culling, for example, testing a BB indicates:

- that all objects beneath it can be removed from further consideration if the BB lies completely outside the view frustum;
- that all objects beneath it need to be displayed if the BB lies completely inside the view frustum;
- or, if the BB intersects the frustum, that the BB for the next level down in the hierarchy needs

to be checked.

An HBB SMS can be created with `mav_SMSHBBNew` (MFS p 128):

```
MAV_SMS *mav_SMSHBBNew(void);
```

The functions which create an SMS, regardless of its type, always return a pointer to the generic SMS type, `MAV_SMS`. Therefore, in theory, to modify any of the example programs to use an HBB SMS requires only that the line creating the SMS be changed. However, in practice making this simple change would not be sufficient since it would require all the objects to be static.

Broadly speaking, the objects in an application can be divided into those which are static, such as the walls and floors of a building, and those which are dynamic, for example a moving avatar. Typically, an application would store the static objects in an HBB SMS, and the dynamic objects in an object list SMS.

Objects can be freely moved between SMSs using the functions `mav_SMSObjectAdd` and `mav_SMSObjectRmv`. For example, if an application wishes to make changes to an object which is normally static, it can remove the object from its HBB SMS and place it in an object list SMS while it undergoes the manipulation. The application can then re-insert it into the HBB SMS, to take advantage of the HBB method's efficiency.

While a hierarchy can be built an object at a time, a more efficient one can be constructed if all the objects which it is to contain are known at the start of construction. Building a HBB SMS in this manner is achieved with the function `mav_HBBConstructFromSMS` (MFS p 284):

```
void mav_HBBConstructFromSMS(MAV_SMS *target, MAV_SMS *from);
```

where `target` is the HBB SMS to construct from the SMS `from`, which can be any of type.

11.2 View modifier functions

As described in Chapter 5.1 (page 47), the MAVERIK viewing model is based on the standard computer graphics viewing model, where the application defines an **eye point**, **view direction vector** and **view up vector**. However, MAVERIK generalises this model by introducing two **view modifier functions**: the per-view modifier function, and the per-window modifier function.

Each view modifier function is used to arbitrarily transform the view parameters, as supplied by the application, to create **modified** view parameters which define the view that is actually used.

The prototype for a view modifer function is:

```
typedef void (*MAV_viewModifierFn)(MAV_window *);
```

11.2.1 The Per-view modifier function

The first view modifier function is the **per-view modifier**, which an application could use to adjust the original eye point to implement an over-the-shoulder view for an avatar, or to modify the view direction so that it tracks the orientation of a head-mounted display.

The per-view modifier function is set using the `mod` field of the `MAV_viewParams` data structure. If defined, this function is executed at the start of the frame to transform the application supplied values of `eye`, `view`, `up` and `right` storing the new values in the `trans_eye`, `trans_view`, `trans_up` and `trans_right` fields of the `MAV_viewParams` data structure.

The `allobjs` example program in the `examples/misc/allobjs` sub-directory of the MAVERIK distribution demonstrates how per-view modifier functions work. Pressing "l" in this example sets the `mod` field of the `MAV_viewParams` data structure to be `lookabout`, pressing "f" sets it back to `NULL`. `lookabout` is implemented as follows:

```
void lookabout(MAV_window *w)
{
  float ax, ay;

  /* Calculate transformed view parameters */
  w->vp->trans_eye= w->vp->eye;
  w->vp->trans_view= w->vp->view;
  w->vp->trans_up= w->vp->up;
  w->vp->trans_right= w->vp->right;

  /* Rotate view parameters by an amount governed by the mouse pos */
  ax= (w->width/2-mav_mouse_x)*0.01;
  ay= (w->height/2-mav_mouse_y)*0.01;

  /* Pitch view by amount ay */
  w->vp->trans_view= mav_vectorRotate(w->vp->trans_view, w->vp->trans_right, ay);
  w->vp->trans_up= mav_vectorRotate(w->vp->trans_up, w->vp->trans_right, ay);

  /* Yaw view by amount ax */
  w->vp->trans_view= mav_vectorRotate(w->vp->trans_view, w->vp->trans_up, ax);
  w->vp->trans_right= mav_vectorRotate(w->vp->trans_right, w->vp->trans_up, ax);
}
```

When activated, `lookabout` is called at the start of each frame and causes the final view to be a pitched and yawed version of the view originally defined by the application. (The viewing parameters are pitched by an amount given by the mouse's vertical position and yawed by an amount given by the mouse's horizontal position).

Note: the modified view **does not overwrite** the application-defined view. A view modifier function takes as input a `MAV_window` data structure from which the relevant `MAV_viewParams` data structure is trivially obtained (from the `vp` field).

The `tdm` example in the `examples/misc/TDM` sub-directory of the MAVERIK distribution demon-

strates how a per-view modifier function can be used to make the view track a 6 DOF tracking device, such as a Polhemus sensor mounted on a head-mounted display. Note: this example will only meaningfully work if MAVERIK has been compiled with the TDM option enabled.

11.2.2 The Per-window modifier function

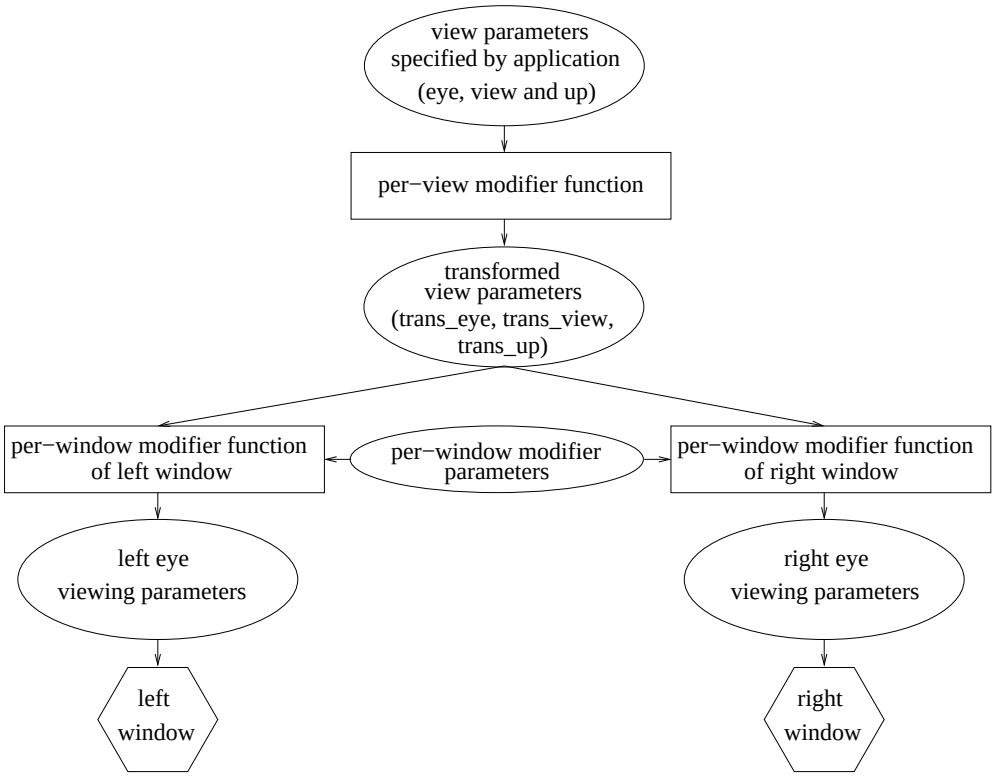
The second view modifier function is the **per-window modifier** which an application can use to generate stereo views.

The per-window modifier function is set using the `mod` field of the `MAV_window` data structure. If defined, this function is is executed at the start of the frame, but **after** the per-view modifier function, to transform the values of `trans_eye`, `trans_view`, `trans_up` and `trans_right` storing the new values in the `eye`, `view`, `up` and `right` fields of the `MAV_window` data structure. It is these transformed values which are used to define the view for the frame.

Stereo viewing is typically achieved by creating two windows which share a common set of `MAV_viewParams` but have different per-window modifier functions – one which offsets the view to the left, the other which offsets the view to the right.

Allied to this calculation is a set of **view modifier parameters** which are common to both windows and contain the information needed to calculate the stereo view, such as the eye offset.

The full path for defining the view is shown below:



Although an application can define the per-window modifier function and parameters for a window directly by setting the relevant fields in the `MAV_window` data structure, MAVERIK provides the `mav_windowViewModifierSet` (MFS p 230) function to perform this task:

```
void mav_windowViewModifierSet(MAV_window *w, MAV_viewModifierParams *vmp,
                               MAV_viewModifierFn fn);
```

The `MAV_viewModifierParams` (MFS p 35) data structure is as follows:

```
typedef struct {
  float offset;
  float angle;
  void *userdef;
} MAV_viewModifierParams;
```

MAVERIK provides two per-window modifier functions, `mav_eyeLeft` (MFS p 138) and `mav_eyeRight` (MFS p 138), which translates `trans_eye` along the `trans_right` vector by $-\text{offset}/2.0$ and $+\text{offset}/2.0$ respectively.

The other fields of the `MAV_viewModifierParams` data structure are currently unused. However, per-window modifier functions could be written which use `angle` to implement a stereo view with convergence or use the `userdef` field to attach arbitrary data.

If `mav_opt_stereo` is used to a open stereo window pair, they are automatically assigned per-window modifier functions as follows:

```
mav_windowViewModifierSet(mav_win_left, &mav_stp_default, mav_eyeLeft);
mav_windowViewModifierSet(mav_win_right, &mav_stp_default, mav_eyeRight);
```

where `mav_stp_default` is the default view modification parameters introduced in Chapter 6.2.3 (page 62).

Part IV

MAVERIK Programming Level 3

Chapter 12

Adding new input devices and modules

Please email maverik@aig.cs.man.ac.uk for advice on this topic.

Chapter 13

Customising spatial management

Please email maverik@aig.cs.man.ac.uk for advice on this topic.

Appendix A

Running MAVERIK applications

A.1 Installing MAVERIK, and compiling with it

For full details see the INSTALL file in the top level directory of the MAVERIK distribution. For your convenience, this is reproduced in Section A.4.

A.1.1 Environment variables

MAVERIK libraries are dynamically linked. Therefore, you may need to set your LD_LIBRARY_PATH (or LD_LIBRARYN32_PATH on Irix6) to point to their location if they are not in one of the standard places. For example,

```
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/maverik-6.1/lib
```

The TDM libraries, if used, are also dynamically loaded and so their location may also need to be present in the LD_LIBRARY_PATH.

Some MAVERIK applications, e.g the avatar example (in examples/misc/avatar), require you to set the MAV_HOME environment variable to point to the MAVERIK distribution. For example,

```
export MAV_HOME=/usr/local/maverik-6.1
```

A.2 A sample Makefile

Here's a simple Makefile for compiling with MAVERIK.

```
CC= gcc
CFLAGS= -I/usr/local/maverik-6.1/incl
LIBS= -L/usr/local/maverik-6.1/lib -lmaverik

eg1:         eg1.o
             ${CC} eg1.o -o eg1 ${LIBS}
```

Feel free to copy the `Makefile` that comes with the MAVERIK distribution examples. To use these, two environment variables need to be set:

1. `MAV_HOME` to indicate where Maverik is installed – see above.
2. `CC` to specify the various compiler options
   e.g. for GNU/Linux

   ```
   export CC="gcc -O2 -finline-functions -fomit-frame-pointer -funroll-loops -ffast-math -march='uname
   ```

   e.g. for Irix6 with an R10k processor

   ```
   export CC="cc -n32 -mips4 -r10000 -O3"
   ```

A.3 Keyboard function keys

MAVERIK recognises the following key presses while an application is running:

- **Shift-Esc** quits the running program
- **Shift-F1** decrease stereo offset by 10%
- **Shift-F2** decrease stereo offset by 1%
- **Shift-F3** increase stereo offset by 1%
- **Shift-F4** increase stereo offset by 10%
- **Shift-F5** swap left and right windows
- **Shift-F6** toggle stereo offset between value and 0
- **Shift-F7** print window and view information to `stdout`
- **Shift-F8** toggle wireframe/filled
- **Shift-F9** toggle multisample (where applicable)
- **Shift-F10** toggle drawing mouse as cross-hairs at world coordinates
- **Shift-F11** dump the window the mouse is in as `snap[n].ppm`

- **Shift-F12** prints status information to `stdout`
- **Ctrl-F2** decrease linear navigation scaling factor by 10%
- **Ctrl-F3** increase linear navigation scaling factor by 10%
- **Ctrl-F4** toggle LOD
- **Ctrl-F5** decrease near clipping plane by 10%
- **Ctrl-F6** increase near clipping plane by 10%
- **Ctrl-F7** decrease far clipping plane by 10%
- **Ctrl-F8** increase far clipping plane by 10%
- **Ctrl-F9** decrease field of view by 10%
- **Ctrl-F10** increase field of view by 10%
- **Ctrl-F12** load a module on the fly

Note that some window managers trap these key presses themselves and so they are not guaranteed to work. This is particularly true for the **Shift-Esc** sequence.

Shift-F1 – **Shift-F6** will only be enabled for stereo applications.

A number of **Ctrl-F** key presses may also be defined depending on how MAVERIK was compiled and the supporting modules present. For example if Voodoo acceleration is active then MAVERIK traps the **Ctrl-F1** keypress to toggle between full-screen and in-window rendering.

A.4 The MAVERIK INSTALL file

```
Building GNU Maverik on UNIX machines
=====================================

To build Maverik you need X11 and either OpenGL, Mesa (version 3.1 or
above) or IrisGL.

Untar the Maverik distribution file which creates the directory
structure shown at the end of this document.

Move into the Maverik directory and type "./setup" followed by
"make". This will compile the Maverik library, example programs, and
any demo programs present.

The "setup" script generates a Makefile appropriate to your
machine. Currently supported UNIX platforms are Irix, RedHat, FreeBSD
and SunOS5. However, we believe that Maverik itself should compile on
```

```
any UNIX platform with X11 and OpenGL/Mesa, so, if you are using a
different platform, edit the script, fill in the appropriate values
and mail us the amendments for inclusion in the next release.

The setup script accepts the following command line options:

--help - prints a help message detailing these arguments.
--debug - to compile with "-g" rather than full optimization.
--VRML97 - to specify that you want VRML97 support (requires a C++
           compiler, flex and bison).
--MESAPATH - to specify where Mesa is installed if it is not somewhere
             where the compiler will automatically pick up. Also see
             note below.
--XLIBPATH - to specify where the X library is installed if not in
             /usr/X11R6/lib.
--IrisGL - to specify that you want an IrisGL based graphics module
           (no longer actively supported).
--GTK - to specify that you want a GTK based graphics module
--D3D - to specify that you want a Direct3D based graphics module
       (Cygwin only and see http://www.jabadaw.co.uk/maverik)
--D3DINCL - to specify the location of the Direct3D include files
--TDMPATH - to specify the use and location of TDM (6 DOF input device
            support).
--TRINCL and --TRLIBS - to specify the use of the Tiled Rendering
                        library and where to find its include and
                        library file.
--PNGINCL and --PNGLIBS - as above but for the PNG library
                          (requires the zlib library)

Not all options are applicable to all platforms.

An Irix6.x complication use the n32 ABI and, if an R10k processor is
detected, mips4 and r10000 options.

"-O2" optimization is used on Irix platforms, and "-O2
-finline-functions -fomit-frame-pointer -funroll-loops -ffast-math -march=‘uname -m‘"
on RedHat and FreeBSD.

If the Mesa library file is not installed somewhere where the compiler
will automatically pick it up, RedHat users may need to add its
location to their LD_LIBRARY_PATH in order to link and run the Maverik
examples. This is not the case for an RPM installation of Mesa.

FreeBSD users: run the setup script and compile as root. When
compilation is complete type: ldconfig -m <Maverik path>/lib
If when compiling your own Maverik program an error message is
produced for including the header files, the easiest solution is to:
mkdir /usr/include/maverik; cp <Maverik path>/incl/* /usr/local/include/maverik/
Any questions on compiling and running Maverik on FreeBSD should be
directed to Joe Topjian (kazar@telerama.com).
```

```
Building GNU Maverik on Windows machines
========================================

There are two broad options for building Maverik on a Windows machine:
(1) use Cygwin, or (2) use a native Windows compiler such as Microsoft
Visual C++ or Borland C++ builder (we do not have access to other
compilers to test the code with).

Note that with both of these options the Maverik libraries are
statically linked and the kernel assumes that (at least) all of the
standard modules are present. Because of this the kernel examples
cannot be compiled.

Using Cygwin
------------

The Cygwin tools are ports of the popular GNU development tools and
utilities for Windows 9x/ME/NT/2000. This provides a UNIX like
environment in which to build Maverik. Version 1.1.6 or newer of
Cygwin is required to compile Maverik. Cygwin can be downloaded for
free from http://www.cygwin.com.

To build Maverik using Cygwin start a bash shell and follow the Unix
instructions given above (although not all of the options to the setup
script are applicable).

More information on installing Cygwin and Maverik can be found at
http://www.jabadaw.co.uk/maverik

Using Visual C++
----------------

Workspace and project files for Microsoft Visual C++ (version 6.0) can
be found in the vc++ sub-directory of the Maverik distribution. These
compile the Maverik libraries and example programs.

Move into the vc++ sub-directory and double-click on "maverik.dsw".
This should launch Visual C++. Build the libraries and examples by
selecting batch build (Build->Batch Build->Build).

The executables for the examples are placed in the same sub-directory
of the Maverik distribution as their respective source files,
eg. examples/MPG, examples/misc/stereo. Some of the examples need to
be executed from this location in order to correctly pick up texture
and model files. Move into the directories and double-click on the
relevant icons. Further, some examples need command line arguments so
simply double-clicking the icon or executing them from Visual C++ will
not work.

The Maverik libraries are automatically built as needed by the
examples and are placed in the lib sub-directory. The Maverik
libraries are built using the "debug single-threaded" runtime
libraries and thus may fail to link with code built using different
```

```
versions of the runtime libraries.

To build the Maverik demos (if present) double click on "demos.dsw".

To build a Direct3D version of Maverik, instead of the default OpenGL,
rename the libmaverikd3d.dsp file in the vc++ sub-directory to be
libmaverik.dsp.

Using Borland C++ Builder
-------------------------

See http://www.jabadaw.co.uk/maverik/borland.html.



Building GNU Maverik on a Macintosh
===================================

To be able to compile Maverik you must install Apple's implementation
of OpenGL on your Mac (http://www.apple.com/opengl). The code has been
tested with the Metrowerks Codewarrior compiler. To compile Maverik
with this compiler: add all files in the src/kernel, src/callbacks,
src/SMS, src/windows, src/navigation, and src/objects directories plus
the src/gfx/mav_gfxOpenGL.c and src/gfx/mav_gfxWMOpenGLMacOS.c files
to a new Codewarrior project; ensure that the "incl" directory is in
the project's access paths, via the Project Settings dialog; and add
the OpenGLStubLib and OpenGLStubUtilLib from Apple's OpenGL SDK to the
project.

By default a hardware accelerated OpenGL context is used, in order to
use software rendering define MAV_MACNOACC during compiling.



Testing the system
==================

To test that the Maverik libraries and examples have been successfully
compiled, we suggest you try executing eg1 in the examples/MPG
sub-directory of the Maverik distribution. You should see a window
appear and display the Maverik welcome message (which consists of
a spiraling Maverik logo with various copyright, version and contact
information). When the message clears you should see an empty blue
window. Press Shift-ESC to quit.

Note that the Maverik libraries are dynamically linked so you may need
to set your LD_LIBRARY_PATH (or LD_LIBRARYN32_PATH for Irix6) to
include the Maverik library directory in order to run the example
program, e.g. for a bash shell:

export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/maverik-6.1/lib

RedHat users may also need to add the location of the Mesa library to
```

```
this line, see note in previous section. The location of the TDM
libraries, if used, may also need to be added to the LD_LIBRARY_PATH
environment variable.



Compiling with Maverik
======================

The Makefiles for the example programs have been made as simple as
possible so as to allow them to be used for compiling your own Maverik
programs (or indeed versions of the examples that have been copied
into a users workspace and modified). To use these Makefiles you
will need to set the following environment variables:

1) MAV_HOME to indicate where Maverik is installed
   e.g. export MAV_HOME=/usr/local/maverik-6.1

2) CC to specify the various compiler options
   e.g. for RedHat
   export CC="gcc -O2 -finline-functions -fomit-frame-pointer -funroll-loops -ffast-math -march=‘uname -m

   e.g. for Irix6 with an R10k processor
   export CC="cc -n32 -mips4 -r10000 -O3"

Both of these environment variables are correctly set for you when the
example are automatically compiled.



Contents
========

The Maverik distribution contains the source code, include files,
worked example programs and documentation consisting of a Programmers
Guide (to accompany the examples) and a Functional Spec. Also
available via a separate download is a Maverik "demos" distribution
which contains a number of applications that have been developed using
Maverik.

The tar file creates the following directory structure:

 maverik-6.1
    |
    +-- demos (essentially empty, available as a separate download)
    |
    +-- doc
    |    |
    |    +-- MPG (Maverik Programmers Guide)
    |    |    |
    |    |    +-- sub-directories for postscript and pdf versions
    |    |
    |    +-- MFS (Maverik Functional Specification)
```

```
|         |
|         +-- sub-directories for postscript, pdf, HTML and man page versions
|
+-- examples
|    |
|    +-- MPG
|    |
|    +-- kernel
|    |
|    +-- misc
|         |
|         +-- various sub-directories
|
+-- incl
|
+-- lib
|
+-- src
|    |
|    +-- SMS
|    |
|    +-- callbacks
|    |
|    +-- extras
|    |    |
|    |    +-- various sub-directories
|    |
|    +-- gfx
|    |
|    +-- kernel
|    |
|    +-- navigation
|    |
|    +-- objects
|    |
|    +-- windows
|
+-- vc++ (workspace and project files for compiling with Visual C++)
```

Appendix B

The default objects

MAVERIK provides the following sets of default objects:

- 15 solid 3D object "primitives": box, pyramid, cylinder, cone, sphere, half sphere, ellipse, half ellipse, circular torus, rectangular torus, polygon, polygon group, facet, rectangle and teapot;
- 2 line-based, rather than solid, objects: the polyline and text. These allow for the definition of objects comprising 3D lines and text respectively;
- 2 "composite" objects, i.e. built from a collection of other objects;

The data structures used to represent these objects are detailed in the following sections. Each of these data structures can act as a starting point for customizing the object.

There are 3 fields common to all the object data structures: `sp`, `matrix` and `userdef`.

- `sp` – specifies which set of surface parameters are to be used to render this object. This is a pointer thus allowing a set of objects to share a single set of parameters. The `MAV_surfaceParams` data structure is described in Chapter 6.1 (page 55).
- `matrix` – a transformation matrix which maps the object's local coordinate system into the world coordinates used by the application, defining its position, orientation and scale.
- `userdef` – a void pointer which can be cast by an application to point to its own data structures. This provides a quick way of adding application specific data structures to a default object without the need for modifying its data structure and source code. The use of this feature is demonstrated in Chapter 5.3 (page 52).

It only make scene for the line-based objects to be rendered with an emissive colour. Using any other colouring scheme for their surface parameters leads to undefined results.

Textures can be mapped onto the surface of a solid object in a number of different ways. The manner in which it is implemented for these objects should be viewed, like the data structures, as something to be customized to fit an application's own needs.

All objects have:

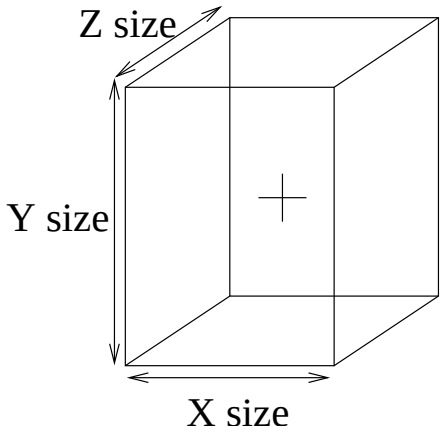
- an object class name `mav_class_*`;
- a data structure of the form `MAV_*`;
- a rendering callback function `mav_*Draw`;
- a bounding box callback function `mav_*BB`;
- an intersection callback function `mav_*Intersection`;
- an identify callback function `mav_*ID`;
- a get matrix callback function `mav_*GetMatrix`;
- a get userdef callback function `mav_*GetUserdef`;
- a get surface parameters callback function `mav_*GetSurfaceParams`;
- a dump callback function `mav_*Dump`.

The source code for the default objects can be found in the `src/objects` sub-directory of the MAVERIK distribution. The code to render the objects is written in MAVERIK's abstracted graphics layer, which is described in Chapter 7.3.3 (page 77) and should be familiar to anyone with a working knowledge of OpenGL.

In the following subsections the diagrams employ a right-handed coordinate system and a cross-hair marks the origin where applicable.

B.1 Box

An axis-aligned box is defined with its center at the origin. It has a dimension, `size`, along the X,Y and Z axis.

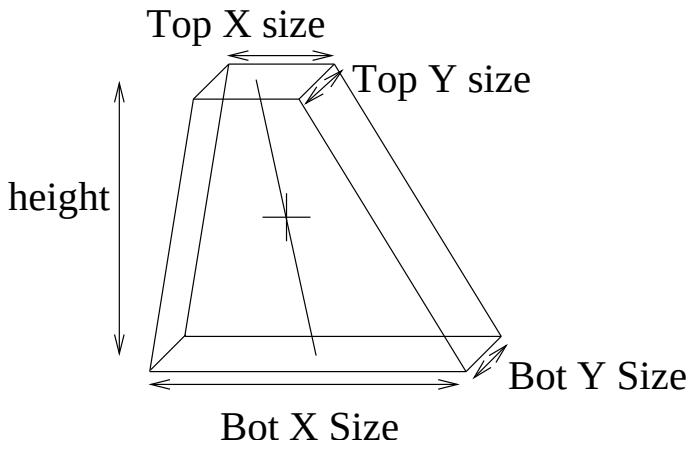


Data structure:

```
typedef struct {
  MAV_vector size;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_box;
```

B.2 Pyramid

A pyramid is defined with its centre at the origin. Its top and bottom faces, which are in the *XY* plane, have a size `[top|bot]_size_[x|y]`. The pyramid has a height, `height`, along the *Z* axis. The *X*,*Y* centres of the top and bottom faces are offset by `offset_x` and `offset_y` respectively.

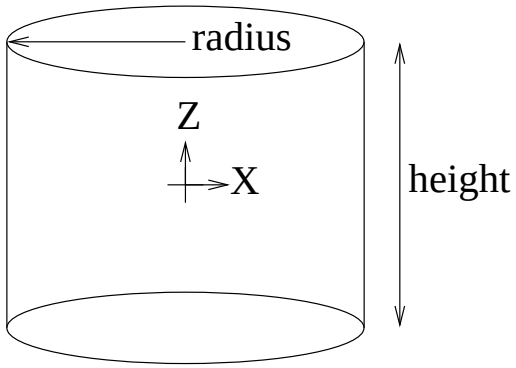


Data structure:

```
typedef struct {
  float bot_size_x;
  float bot_size_y;
  float top_size_x;
  float top_size_y;
  float offset_x;
  float offset_y;
  float height;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_pyramid;
```

B.3 Cylinder

The cylinder is defined with its centre at the origin and its axis aligned along the *Z* axis. It has a radius, `radius`, and a height, `height`, along the *Z* axis.



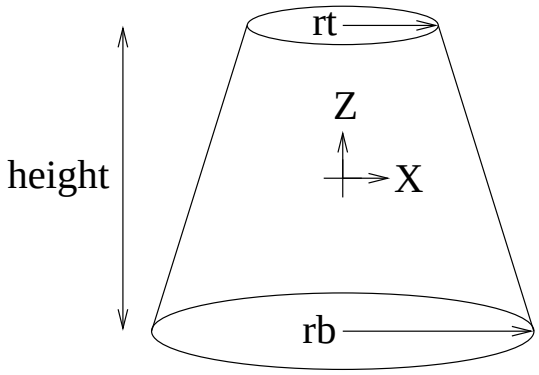
When rendered, `nverts` vertices are used (if `mav_opt_curveLOD` is not set) to facet the curved surface of the cylinder, and the symbolic constant `endcap`, set to `MAV_TRUE` or `MAV_FALSE`, control whether or not the object has endfaces or is effectively hollow.

Data structure:

```
typedef struct {
  float radius;
  float height;
  int nverts;
  int endcap;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_cylinder;
```

B.4 Cone

The cone is defined with its centre at the origin and its axis aligned along the *Z* axis. It has a radius at its top, `rt`, a radius at its bottom, `rb`, and a height, `height`, along the *Z* axis.



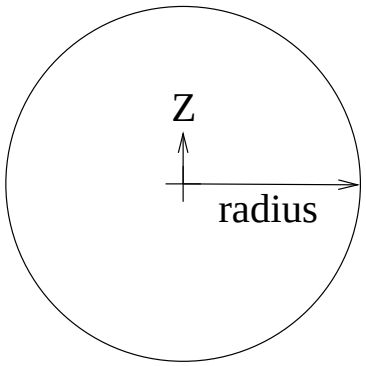
When rendered, `nverts` vertices are used (if `mav_opt_curveLOD` is not set) to facet the curved surface of the cone, and the symbolic constant `endcap`, set to `MAV_TRUE` or `MAV_FALSE`, control whether or not the object has endfaces or is effectively hollow.

Data structure:

```
typedef struct {
  float rt;
  float rb;
  float height;
  int nverts;
  int endcap;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_cone;
```

B.5 Sphere

A sphere is defined with its centre at the origin with a radius `radius`.



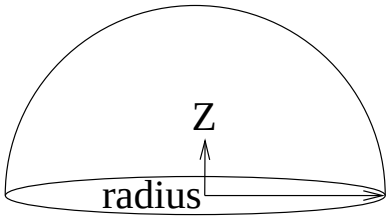
When rendered, `nverts` vertices are used to facet the curved surface of the sphere around the *Z* axis, and `nchips` vertices around the *X* axis from -90 to 90 degrees. Both values are only applicable if `mav_opt_curveLOD` is not set.

Data structure:

```
typedef struct {
  float radius;
  int nverts;
  int nchips;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_sphere;
```

B.6 Half sphere

The half sphere is defined as the positive *Z* half-space of a sphere.



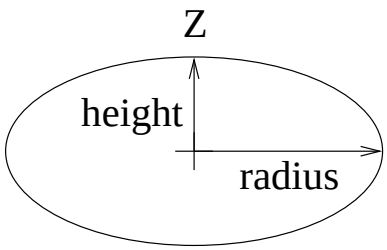
When rendered, `nverts` vertices are used to facet the curved surface of the half sphere around the *Z* axis, and `nchips` vertices around the *X* axis from 0 to 90 degrees. Both values are only applicable if `mav_opt_curveLOD` is not set. The symbolic constant `endcap`, set to `MAV_TRUE` or `MAV_FALSE`, control whether or not the object has an endface or is effectively hollow.

Data structure:

```
typedef struct {
  float radius;
  int nverts;
  int nchips;
  int endcap;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_hsphere;
```

B.7 Ellipse

An ellipse is defined with its centre at the origin and with a radius, `height`, along the *Z* axis and a radius, `radius`, in the *XY* plane.



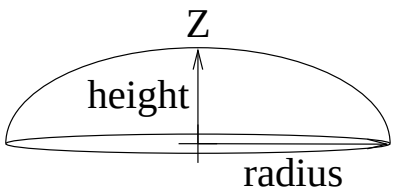
When rendered, `nverts` vertices are used to facet the curved surface of the ellipse around the *Z* axis, and `nchips` vertices around the *X* axis from -90 to 90 degrees. Both values are only applicable if `mav_opt_curveLOD` is not set.

Data structure:

```
typedef struct {
  float radius;
  float height;
  int nverts;
  int nchips;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_ellipse;
```

B.8 Half ellipse

The half ellipse is defined as the positive *Z* half-space of an ellipse.



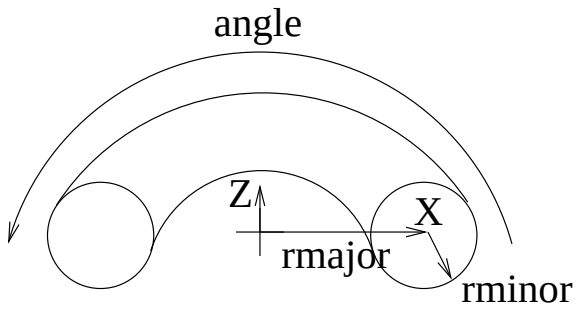
When rendered, `nverts` vertices are used to facet the curved surface of the half ellipse around the *Z* axis, and `nchips` vertices around the *X* axis from 0 to 90 degrees. Both values are only applicable if `mav_opt_curveLOD` is not set. The symbolic constant `endcap`, set to `MAV_TRUE` or `MAV_FALSE`, control whether or not the object has an endface or is effectively hollow.

Data structure:

```
typedef struct {
  float radius;
  float height;
  int nverts;
  int nchips;
  int endcap;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_hellipse;
```

B.9 Circular torus

The circular torus (a torus with a circular cross section) is defined with its centre at the origin and with a major radius, `rmajor`, a minor radius, `rminor`, and to an angular extent, `angle`, in radians from the *X* axis around the *Z* axis.



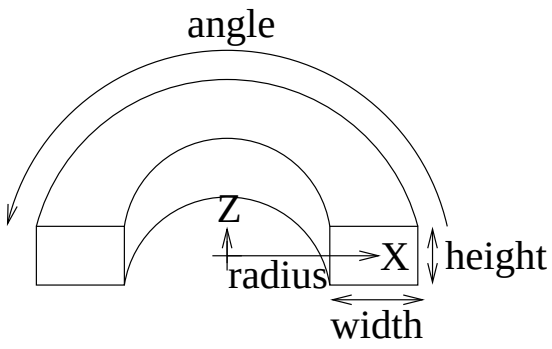
When rendered, `nverts` vertices are used to facet the curved surface defined by the minor radius, and `nchips` vertices the curved surface defined by the major radius. Both values are only applicable if `mav_opt_curveLOD` is not set. The symbolic constant `endcap`, set to `MAV_TRUE` or `MAV_FALSE`, control whether or not the object has endfaces or is effectively hollow.

Data structure:

```
typedef struct {
  float rmajor;
  float rminor;
  float angle;
  int nverts;
  int nchips;
  int endcap;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_ctorus;
```

B.10 Rectangular torus

The rectangular torus (a torus with a rectangular cross section) is defined with the centre at the origin and with a radius, `radius`, a height, `height`, width, `width` and to an angular extent, `angle`, in radians from the *X* axis around the *Z* axis.



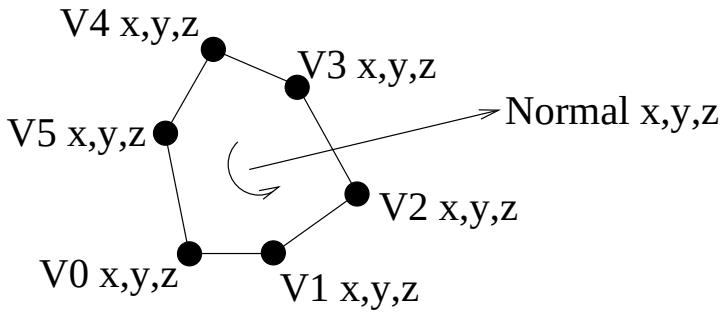
When rendered, `nchips` vertices are used (if `mav_opt_curveLOD` is not set) to facet the curved surface defined by the radius. The symbolic constant `endcap`, set to `MAV_TRUE` or `MAV_FALSE`, control whether or not the object has endfaces or is effectively hollow.

Data structure:

```
typedef struct {
  float radius;
  float width;
  float height;
  float angle;
  int nchips;
  int endcap;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_rtorus;
```

B.11 Polygon

A polygon is defined by a number, `np`, of points, a normal, `norm`, and collection of vertices, `vert`, and, optionally, texture coordinates, `tex`. The polygon must be convex, planar and the vertices ordered anti-clockwise around the normal.



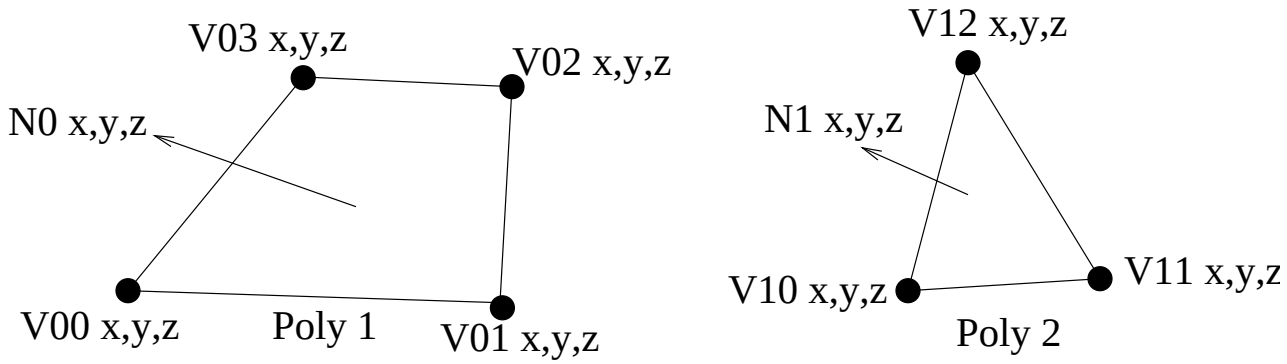
Texture coordinates must be provided is this object is to be textured.

Data structure:

```
typedef struct {
  int np;
  MAV_vector norm;
  MAV_texCoord *tex;
  MAV_vector *vert;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_polygon;
```

B.12 Polygon group

A polygon group is a number, `npolys`, of polygons, each defined as above, which share a common transformation matrix. Polygon groups can be used to define objects which comprise of many polygons without the rendering inefficiency of each polygon having an individual transformation matrix.

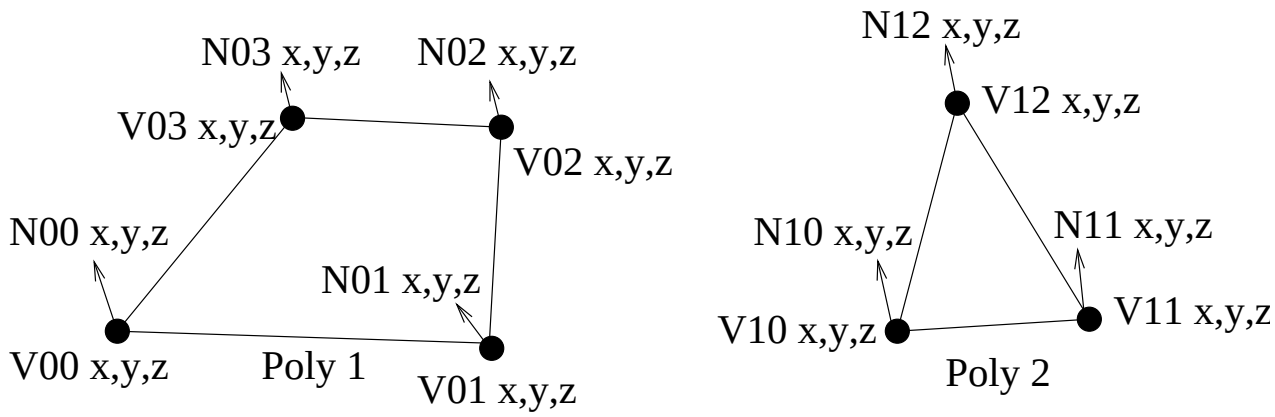


Data structure:

```
typedef struct {
  int npolys;
  int *np;
  MAV_vector *norm;
  MAV_texCoord **tex;
  MAV_vector **vert;
  MAV_surfaceParams **sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_polygonGrp;
```

B.13 Facet

A facet is a number of polygons which share a common transformation matrix and which allow a normal to be defined for each vertex, rather than for each polygon, thus allowing Gouraud shading across the face of the polygon. They are defined in a similar manner to the polygon group, but with a normal, `norm`, per vertex.

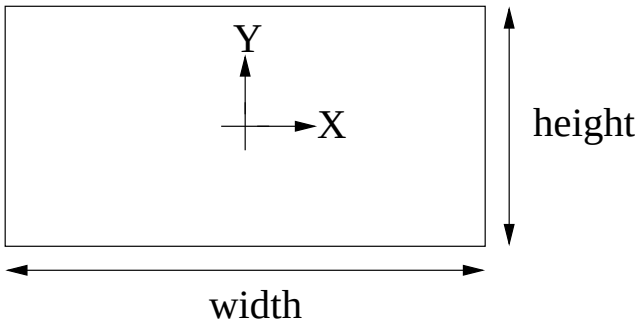


Data structure:

```
typedef struct {
  int npolys;
  int *np;
  MAV_vector **norm;
  MAV_texCoord **tex;
  MAV_vector **vert;
  MAV_surfaceParams **sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_facet;
```

B.14 Rectangle

The rectangle allows for a simple definition of the common case of a 4-vertex polygon centred at the origin with its normal aligned along the positive *Z* axis. Is is defined by its `width` and `height` along the *X* and *Y* axis respectively.



If textured, it is tiled a number of time in the horizontal, `xtile`, and vertical, `ytile`, directions.

Data structure:

```
typedef struct {
  float width;
  float height;
  float xtile;
  float ytile;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_rectangle;
```

B.15 Teapot

The classic computer graphics teapot without which no Virtual Environment is complete. The teapot is orientated with its *Y* axis as "up" and the spout pointing along the positive *X* axis. The teapot has an extent `size` between the edge of the handle and the tip of the spout. The bezier surfaces by which the teapot is defined are subdivided `subdivisions` times when rendered (if `mav_opt_curveLOD` is not set).

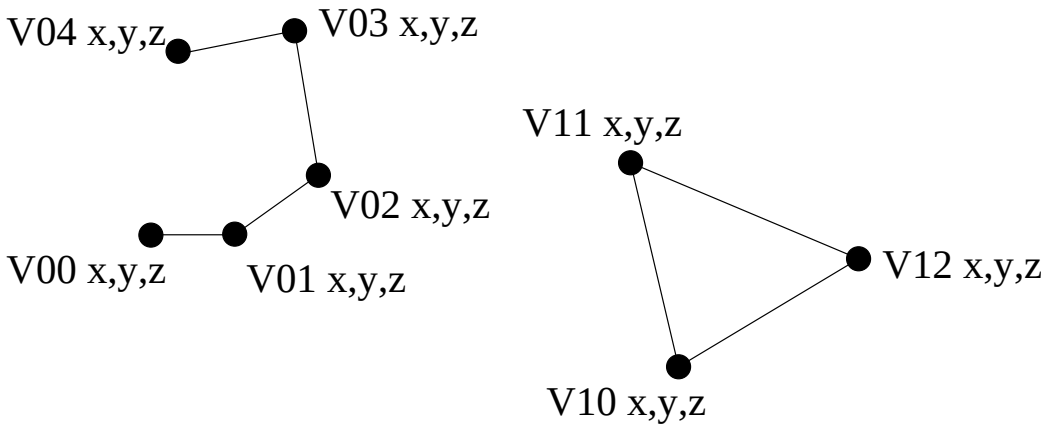
The type of tea used to brew-up is governed by the enumerated constant `teabag` which is set to either `TETLEY`, `PG_TIPS` or `EARL_GREY`. The amount of sugar used is controlled by `lumps`, in units of heaped teaspoons, and should be set to less than 2 otherwise you'll get fat and rot your teeth.

Data structure:

```
typedef struct {
  float size;
  int subdivisions;
  MAV_teabag teabag;
  int lumps;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_teapot;
```

B.16 Polyline

A polyline is a number, `nlines`, of lines each consisting of a number, `np`, of vertices, `vert`, each connected by a line. `closed` indicates if the last vertex connects back to the first.



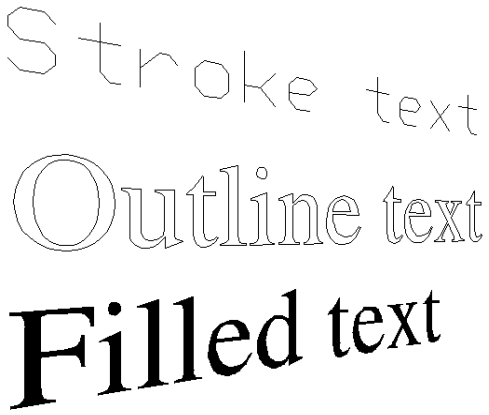
Since it only make sense for this object to be rendered with an emissive colour, attempting to render it with a material or texture gives undefined results.

Data structure:

```
typedef struct {
  int nlines;
  int *np;
  int *closed;
  MAV_vector **vert;
  MAV_surfaceParams **sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_polyline;
```

B.17 Text

The text object allows 3D text to be rendered in a scene. The text, `text`, is defined in the *XY* plane with the tallest character being approximately 1 unit along the *Y* axis. The origin is halfway up the text and, depending on the value of `justify`, set to `MAV_[LEFT|CENTRE|RIGHT]_JUSTIFY`, is either at the left edge of the text, at its centre or at the right edge respectively. `style` can take the value `MAV_[STROKE|OUTLINE|FILLED]_FONT`, to produce the fixed point and proportional styles of text shown below.



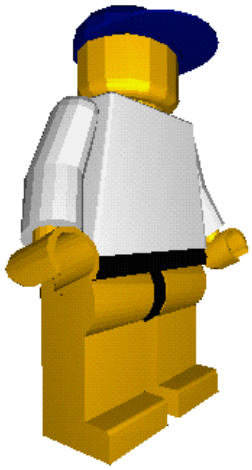
Data structure:

```
typedef struct {
  char *text;
  int style;
  int justify;
  MAV_surfaceParams *sp;
  MAV_matrix matrix;
  void *userdef;
} MAV_text;
```

B.18 Composite object

A composite object is a number, `numobj`, of objects which are first transformed by a common transformation matrix, `matrix`, before being transformed by their individual transformation matrices. The objects are defined as an array, `obj`, of pointers to MAVERIK objects. Once defined, the objects comprising the composite object must remain static, i.e. changing the number of objects in it, or any details of those objects, is forbidden. And since the contents are static, a local coordinate frame bounding box is stored in `bb` for efficiency.

Composite objects are not intended to be defined directly by an application, but rather by routines such as `mav_compositeRead`, which defines a composite object from a VRML97, Lightwave or AC3D format file (as shown below).



If a composite object is selected via the usual mechanism then the integer `selobj` holds the array element of the selected sub-object.

Data structure:

```
typedef struct {
  int numobj;
  MAV_object **obj;
  MAV_BB bb;
  int selobj;
  char *filename;
  MAV_matrix matrix;
  void *userdef;
} MAV_composite;
```

B.19 SMS object

An SMS object contains an SMS, `sms`, of objects which are first transformed by a common transformation matrix, `matrix`, before being transformed by their individual transformation matrix. Objects can be freely added to and removed from the SMS using the usual functions for manipulating SMS's.

An SMS object can be added as an object to any other SMS object, enabling hierarchical structures to be constructed.

If this object is selected via the usual mechanism then `selobj` holds a pointer to the selected subobject.

Data structure:

```
typedef struct {
  MAV_SMS *sms;
  MAV_object *selobj;
  MAV_matrix matrix;
  void *userdef;
} MAV_SMSObj;
```

Appendix C

MAVERIK global variables

There are three categories of global variables:

- **options** variables: these control MAVERIK's initialisation and subsequent behaviour.
- **information** variables: these reflect the current state of MAVERIK.
- **classes and callbacks** variables: these act as identifiers to various MAVERIK functions.

Application's can only change the values of the first set of global variables – the options – the other two sets should be treated as read only. Furthermore, if an application wishes to change the default value of an options variable, it must do so prior to calling `mav_initialise` unless otherwise stated.

C.1 Options variables

The following variables controls options for the **kernel**, **windows**, **stereo display**, **graphics**, and **objects**.

C.1.1 Kernel options

- `int mav_opt_output` – controls the amount of informational messages written to `stdout`. If `MAV_VERBOSE`, all messages are printed; if `MAV_SILENT`, no messages are printed. Default value: `MAV_VERBOSE`. This variable can be modified at anytime.
- `int mav_opt_objectTables` – if `MAV_TRUE`, indicates that object tables are to be used. Object tables allow for efficient conversion between application objects and their corresponding MAVERIK objects, and also keep track of which SMSs objects are in (allowing deleted objects to be automatically removed from their SMSs). Object tables require memory and applications which are very short of memory may wish to disable object tables by setting this variable to

MAV_FALSE. However, this is recommended only for experienced users familiar with the internal workings of MAVERIK. Default value: MAV_TRUE.

- int mav_opt_fixedRnd – if MAV_TRUE, indicates that the pseudo-random sequence returned by the mav_random function should be from a fixed set of 5000 random numbers, rather than calling drand48. This is useful in debugging applications across platforms since the result of drand48 is platform-dependent. Default value: MAV_FALSE. This variable can be modified at anytime.
- int mav_opt_maxColours – the maximum number of emissive colours available in a palette. Default value: 150.
- int mav_opt_maxMaterials – the maximum number of materials available in a palette. Default value: 150.
- int mav_opt_maxTextures – the maximum number of textures available in a palette. Default value: 150.
- int mav_opt_maxFonts – the maximum number of fonts available in a palette. Default value: 10.
- int mav_opt_maxLights – the maximum number of lights available in a palette. Default value: 5.
- int mav_opt_paletteWarn – whether or not to generate a warning if the contents of a palette are redefined. Default value: MAV_TRUE. This variable can be modified at anytime.

C.1.2 Window control options

These options determine the size and placement of the window opened by mav_initialise. See also Section C.1.3 for how these options relate to a pair of windows for stereo viewing.

- int mav_opt_noWins – if MAV_TRUE, do not open a window. Default value: MAV_FALSE.
- int mav_opt_fullscreen – if MAV_TRUE, open the window fullscreen. Default value: MAV_FALSE.
- int mav_opt_x – x position of left-hand edge of window. Default value: left-hand edge of screen.
- int mav_opt_y – y position of top edge of window. Default value: half way up screen (or top of screen if Voodoo acceleration is active).
- int mav_opt_width – width of window. Default value: half screen size (or 640 if Voodoo acceleration is active).
- int mav_opt_height – height of window. Default value: half screen height (or 480 if Voodoo acceleration is active).
- char *mav_opt_name – name to be placed in window title bar. Default value: name of executable (which is followed by “left” for a stereo configuration).

- `char *mav_opt_disp` – name of X server on which to open window. Default value: `NULL` (the `DISPLAY` environment variable is used).

Note: honouring the requested window size, placement and title bar status is at the discretion of the window manager.

The following variables control the graphical context of a window opened by `mav_windowNew` (and hence by `mav_initialise`):

- `int mav_opt_singleBuf` - requests a single buffered context. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_quadBuf` - requests a quad buffered context. This is one way of generating stereo output and is typically employed with LCD shutter glasses. If a windows is opened with a quad-buffered context requested, any un-allocated right buffers in existing contexts are used before a new window and context are created. Acceptable true values are `MAV_STEREO_QUAD_BUFFERS` and `MAV_STEREO_QUAD_BUFFERS_SEPARATE_Z` (for machines which have separate depth buffers). Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_multiSample` - requests a multisampled context. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_accumBuf` - requests an accumulation buffered context. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_stencilBuf` - requests a stencil buffered context. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_destAlpha` - requests a destination alpha buffered context. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_shareContexts` - requests that multiple contexts share a common set of display lists, textures etc. The first window opened is the "master" context and as such cannot be deleted. Default value: `MAV_TRUE`. This variable can be modified at anytime.
- `int mav_opt_WMPlacement` - if `MAV_TRUE`, let the window manager choose the position of the window. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_restrictMouse` - if `MAV_TRUE`, then prevent the mouse from moving outside the window. Default value: `MAV_FALSE` (or `MAV_TRUE` if Voodoo acceleration is active). This variable can be modified at anytime.
- `int mav_opt_flush` - requests that a `glFlush` be performed before swapping the buffers. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_finish` - requests that a `glFinish` be performed before swapping the buffers. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_syncSwap` - requests that swapping buffers in multiple windows is synchronized (if supported by the hardware). Default value: `MAV_FALSE`.

C.1.3 Stereo configuration options

There are basically two ways of achieving stereo output (see Section 6.2.3, page 62). But however it's achieved, it's helpful to think of having have two separate windows (even if these in fact map onto one physical display area). The following variables control stereo viewing:

- `int mav_opt_stereo` – if set to something other than `MAV_FALSE`, this variable indicates that a pair of windows should be opened. Supported stereo configuration are `MAV_STEREO_TWO_WINS`, `MAV_STEREO_QUAD_BUFFERS` and `MAV_STEREO_QUAD_BUFFERS_SEPARATE_Z`. Default value: `MAV_FALSE`.
- `int mav_opt_right_x` – x position of left-hand edge of right-hand window. Default value: half width of screen.
- `int mav_opt_right_y` – y position of top edge of right-hand window. Default value: half way up screen (or top of screen if Voodoo acceleration is active).
- `int mav_opt_right_width` – width of right-hand window. Default value: half screen width (or 640 if Voodoo acceleration is active).
- `int mav_opt_right_height` – height of right-hand window. Default value: half screen height (or 480 if Voodoo acceleration is active).
- `char *mav_opt_right_name` – name for right-hand window titlebar. Default value: name of the executable followed by "right".
- `char *mav_opt_right_disp` – name of X server on which to open right-hand window. Default value: `NULL` (the `DISPLAY` environment variable is used).

C.1.4 Graphics options

The following variables set various graphics rendering options:

- `int mav_opt_trans` – if `MAV_TRUE`, MAVERIK attempts to correctly deal with transparent objects. This is achieved by detecting the objects which are transparent and rendering them in depth sorted order at the end of the frame. Default value: `MAV_FALSE`. This variable can be modified at anytime.

  An example program showing the use of transparent objects is given in the `examples/misc/transparent` sub-directory of the MAVERIK distribution.
- `int mav_opt_delayTexture` – if `MAV_TRUE`, MAVERIK attempts to efficiently deal with textured objects. This is achieved by rendering all textured objects at the end of a frame, potentially minimizing the context changes made by the graphics pipe and so increasing performance. Default value: `MAV_FALSE`. This variable can be modified at anytime.
- `int mav_opt_bindTextures` – if `MAV_TRUE`, indicates that texture maps should be "bound" for efficiency. For details, see the OpenGL documentation. Default value: `MAV_TRUE`.

- int mav_opt_texComps – the number of colour components that OpenGL will use for the internal storage of textures. For details, see the OpenGL documentation. A value of 1 indicates that a texture will be stored internally as a luminance map, a value of 3 indicates that RGB values will be stored and a value of 4, the default, indicates RGBA values will be stored. Use a value of 3 if no alpha component is needed and texture memory is scarce or the number of bits to represent each texel is limited (for example the Voodoo chipset which is 16 bit).

- int mav_opt_mipmapping – if MAV_TRUE, indicates that texture maps should be mipmapped. Default value: MAV_FALSE.

  Note: mav_paletteTextureMipmappingSet can be used to override the global default for individual textures. An example program showing the use of mipmapping is given in the examples/misc/textures sub-directory of the MAVERIK distribution.

- int mav_opt_trackMatrix – indicates if changes to the model view graphics matrix stack should be tracked and stored in the MAV_window data structure. Recognised values for this variable are:

  – MAV_FALSE – no tracking.
  – MAV_TRUE – track model view changes and store in viewMat field of MAV_window.
  – MAV_PROJANDVIEW – as MAV_TRUE, but also multiplies the current view by the current projection matrix and stores the result in the pdvMat field of MAV_window.

  If this data is frequently used by the application, it is more efficient to track changes in the matrix rather than repeatedly requesting its current value with mav_gfxMatrixGet. Default value: MAV_FALSE. This variable can be modified at anytime.

C.1.5 Object options

The following variables set options which control the behaviour of the default objects:

- int mav_opt_BBMethod – indicates which bounding box callback should be registered for the default objects. Setting to MAV_BB_FAST selects a fast but potentially pessimistic bounding box calculation; setting to MAV_BB_ACCURATE selects a slower but more accurate calculation. Obviously, these terms are relative. Default value: MAV_BB_ACCURATE.

- int mav_opt_compositeSetMatrix – if MAV_TRUE, indicates that the mav_compositeRead family of functions should set a composite object's matrix to MAV_ID_MATRIX. Default value: MAV_TRUE. This variable can be modified at anytime.

- int mav_opt_curveLOD – if MAV_TRUE, indicates that level-of-detail (LOD) should be performed on objects with curved surfaces, i.e. MAVERIK will ignore the nverts and/or nchips values and render the object with as many, or as few, facets as it deems necessary to accurately represent the curved surface. Default value: MAV_FALSE. This variable can be modified at anytime.

- int mav_opt_vertsMin – the minimum number of vertices that will be used to render an object undergoing LOD processing. Default value: 6. This variable can be modified at anytime.

- `int mav_opt_vertsMax` – the maximum number of vertices that will be used to render an object undergoing LOD processing. Default value: 20. This variable can be modified at anytime.
- `float mav_opt_curveFactor` – an arbitrary constant controlling the rate at which the number of vertices that will be used to render an object undergoing LOD processing is reduced as the object recedes from the eye point. Default value: 10000. This variable can be modified at anytime.
- `float mav_opt_drawNormals` – if set to a value of less than 1000000, then draw the normals of the polygons which comprise the following default objects: facet, polygon group, rectangle and polygon. The value of this variable defines the magnitude of the displayed normal vector. Displaying a polygon's normal can help identify lighting problems. Default value: `MAV_INFINITY`. This variable can be modified at anytime.
- `int mav_opt_VRML97HBBThreshold` – the number of objects above which an HBB SMS, rather than an object list SMS, is used to hold the objects in a VRML97 file. Default value: 0 (always use a HBB). This variable can be modified at anytime.
- `int mav_opt_VRML97CleanUp` – if `MAV_TRUE`, then clean up the data structures used to parse a VRML97 file. This option only exists to overcome a bug in the VRML97 parser which sometimes occurs for large models. Default value: `MAV_TRUE`. This variable can be modified at anytime.

C.1.6 Miscellaneous options

The following variables describe various miscellaneous options:

- `int mav_opt_navPassEvents` – controls whether or not keyboard and mouse navigation events are also passed on to application defined event handling callbacks where applicable. Default value: `MAV_TRUE` (pass on events where applicable). This variable can be modified at anytime.
- `char * mav_opt_TDMLib` – defines the library to dynamically load to provide the implementation of the TDM interface. Default value: `NULL` (prompt user – there is no meaningful default).

C.2 Information variables

The following variables are maintained by MAVERIK, and should be treated as **read-only**. If an application changes the value of any of these variables, the results will be unpredicatble.

C.2.1 Window information

- `MAV_window *mav_win_mono` – the handle to the single window in a mono configuration, or the left window in a stereo configuration.

- MAV_window *mav_win_left – same as mav_win_mono.
- MAV_window *mav_win_right – the handle to the right window in a stereo configuration. Undefined for a mono configuration.
- MAV_window *mav_win_all – a handle to all windows. Setting parameters, such as callbacks and perspective, using mav_win_all will effect all open windows.
- MAV_window *mav_win_current – a handle to the current active rendering window.
- MAV_list *mav_win_list – a MAV_list containing the handles of all open windows.

C.2.2 Frame information

- float mav_fps – the current frame rate (expressed as frames per second). This is computed from the elpased wallclock time between the most recent calls to mav_frameBegin and mav_frameEnd.
- float mav_fps_avg – the frame rate (expressed as frames per second) averaged over the last second.
- int mav_firstFrame – MAV_TRUE only for the very first frame.
- int mav_frameCount – a counter incremented at the end of each frame.
- int mav_drawingMouse – MAV_TRUE if a mouse cursor is being drawn by MAVERIK as a cross.
- int mav_navigating – the number of default navigators currently active.
- int mav_needFrameDraw – the logical OR of mav_firstFrame, mav_drawingMouse and mav_navigating. A "true" value indicates that a frame needs to be drawn even though an event hasn't occurred.

C.2.3 Mouse information

- MAV_window *mav_win_mouse – the handle of the window the mouse is currently in.
- int mav_mouse_x – the mouse's horizontal position in pixels relative to the origin of the window it's in.
- int mav_mouse_y – the mouse's vertical position in pixels relative to the origin of the window it's in.
- int mav_mouse_root_x – as for mav_mouse_x but relative to the root window.
- int mav_mouse_root_y – as for mav_mouse_y but relative to the root window.
- int * mav_mouse_button – an array holding the status (MAV_PRESSED or MAV_RELEASED) of each mouse button. Currently this is only supported on Unix machines. Valid array indices are: MAV_[LEFT|MIDDLE|RIGHT]_BUTTON.

The raw mouse information above is calculated as part of the mouse device poll function, and is updated once per frame in `mav_frameBegin`.

The following mouse information can only be calculated once an eyepoint has been defined and is therefore not available to functions registered with `mav_frameFn1Add` (see `mav_frameBegin`).

- `MAV_vector mav_mouse_pos` – the 3D position of the mouse when mapped onto a plane parallel to the near clip plane but at twice the near clip plane distance. The reason that the near clip plane itself is not used is that it would rule out anything being rendered at this position.
- `MAV_vector mav_mouse_dir` – the normalized vector from the current eye position through `mav_mouse_pos`.

C.2.4 Graphics information

The following variables are defined by the OpenGL implementation being used:

- `char * mav_gfx_vendor` – see OpenGL documentation for description.
- `char * mav_gfx_renderer` – see OpenGL documentation for description.
- `char * mav_gfx_version` – see OpenGL documentation for description.

Note: a valid OpenGL context has to be created, i.e. a window opened, before their value is set.

C.2.5 Miscellaneous information

- `int mav_xres` – the horizontal resolution of the screen in pixels.
- `int mav_yres` – the vertical resolution of the screen in pixels.
- `int mav_this_version` – the value of `MAV_THIS_VERSION`, which is defined in the header file and is unique to a given version, when the library was compiled. This variable can be checked at run-time against the value of `MAV_THIS_VERSION` to ensure that an application is compiled with and linked against the same version of MAVERIK. The variable can also be checked against the values of the `MAV_VERSION` macro to determine which version of the library the application is linked with.
- `MAV_list *mav_module_list` – a `MAV_list` containing the identify function for all registered modules.
- `MAV_list *mav_sms_list` – a `MAV_list` containing all SMS's.
- `MAV_list *mav_palette_list` – a `MAV_list` containing all palettes.

- MAV_list *mav_object_list – a MAV_list containing all registered objects (requires mav_opt_objectTables to be MAV_TRUE). Note that this list will contain the system defined objects mav_object_world, mav_object_any, and mav_object_none.
- MAV_list *mav_transObjList – a MAV_list maintained on a per-frame basis for storing information to allow transparent objects to be rendered correctly.
- MAV_palette *mav_palette_default – the palette windows are associated with until reassigned with mav_windowPaletteSet. Unlike mav_win_all, values set for this palette do not act on all palettes.
- MAV_surfaceParams *mav_sp_current – the current surface parameters.
- MAV_surfaceParams *mav_sp_default – the default surface parameters (a red material).

The following numeric values are, by necessity, implemented as constant global variables rather than #defines.

- const MAV_vector MAV_NULL_VECTOR – the vector (0,0,0).
- const MAV_vector MAV_X_VECTOR – the vector (1,0,0).
- const MAV_vector MAV_Y_VECTOR – the vector (0,1,0).
- const MAV_vector MAV_Z_VECTOR – the vector (0,0,1).
- const MAV_matrix MAV_ID_MATRIX – the identity matrix.
- const MAV_quaternion MAV_ID_QUATERNION – the identity quaternion [1,(0,0,0)].

C.3 Class and callback variables

C.3.1 Object classes

The following object classes are defined on initialisation to represent the default shapes. Each of the object classes are fully described in Appendix B.

- MAV_class *mav_class_box – a handle to the box object.
- MAV_class *mav_class_pyramid – a handle to the pyramid object.
- MAV_class *mav_class_cylinder – a handle to the cylinder object.
- MAV_class *mav_class_cone – a handle to the cone object.
- MAV_class *mav_class_sphere – a handle to the sphere object.
- MAV_class *mav_class_hsphere – a handle to the half sphere object.

- `MAV_class *mav_class_ellipse` – a handle to the ellipse object.
- `MAV_class *mav_class_hellipse` – a handle to the half ellipse object.
- `MAV_class *mav_class_ctorus` – a handle to circular cross-section torus object.
- `MAV_class *mav_class_rtorus` – a handle to the rectangular cross-section torus object.
- `MAV_class *mav_class_polygon` – a handle to the polygon object.
- `MAV_class *mav_class_polygonGrp` – a handle to the polygon group object.
- `MAV_class *mav_class_facet` – a handle to the facet object.
- `MAV_class *mav_class_rectangle` – a handle to the rectangle object.
- `MAV_class *mav_class_polyline` – a handle to the polyline object.
- `MAV_class *mav_class_text` – a handle to the text objet.
- `MAV_class *mav_class_composite` – a handle to the composite object.
- `MAV_class *mav_class_SMSObj` – a handle to the SMS object.
- `MAV_class *mav_class_all` – a handle to all objects.

Callbacks set for `mav_class_all` take precedence over one defined for a specific object class.

C.3.2 Miscellaneous classes

The following classes are defined on initialisation to enable event-based callbacks to be defined which do not occur on a per-object class basis.

- `MAV_class *mav_class_world` – to trap events regardless of where the mouse is pointing.
- `MAV_class *mav_class_any` – to trap events which occur when the mouse is pointing at an object, but regardless of the class of the object.
- `MAV_class *mav_class_none` – to trap events which occur when the mouse is not pointing at any object.

The MAVERIK objects passed to the event callback function set for one of the above classes are respectively `mav_object_world`, `mav_object_any`, and `mav_object_none`. No attempt should be made to interpret the data portion of these MAVERIK objects.

Events set for `mav_class_world` take precedence over those set for `mav_class_any`, which in turn take precedence over those set for a particular object class.

C.3.3 Object-based callback functions

These callback functions are defined on initialisation to act on objects.

- `MAV_callback *mav_callback_delete` – a handle to the "delete" callback.
- `MAV_callback *mav_callback_draw` – a handle to the "draw" callback.
- `MAV_callback *mav_callback_BB` – a handle to the "calculate bounding box" callback.
- `MAV_callback *mav_callback_intersect` – a handle to the "object-line intersection test" callback.
- `MAV_callback *mav_callback_id` – a handle to the "identify" callback.
- `MAV_callback *mav_callback_dump` – a handle to the "dump" callback.
- `MAV_callback *mav_callback_getUserdef` – a handle to the "get userdef" callback.
- `MAV_callback *mav_callback_getMatrix` – a handle to the "get matrix" callback.
- `MAV_callback *mav_callback_getSurfaceParams` – a handle to the "get surface parameters" callback.

Note that any callback defined for `mav_class_all` takes precedence over one defined for a specific object class.

C.3.4 Event-based callback functions

These callback functions are defined on initialisation to trap events.

- `MAV_callback *mav_callback_keyboard` – a handle to the keyboard callback.
- `MAV_callback *mav_callback_sysKeyboard` – a handle to the system reserved keyboard callback.
- `MAV_callback *mav_callback_leftButton` – a handle to the left mouse button callback.
- `MAV_callback *mav_callback_middleButton` – a handle to the middle mouse button callback.
- `MAV_callback *mav_callback_rightButton` – a handle to the right mouse button callback.
- `MAV_callback *mav_callback_anyButton` – a handle to the any mouse button callback.
- `MAV_callback *mav_callback_sysMouse` – a handle to the system reserved mouse callback.
- `MAV_callback *mav_callback_resize` – a handle to the resize callback.

- `MAV_callback *mav_callback_map` – a handle to the mapping callback.
- `MAV_callback *mav_callback_crossing` – a handle to the crossing callback.
- `MAV_callback *mav_callback_expose` – a handle to the expose callback.

A callback set for `mav_callback_anyButton` will take precedence over one set for a particular button.

C.3.5 SMS classes

The following SMS classes are defined on initialisation.

- `MAV_SMSClass *mav_SMSClass_objList` – the simple linked-list SMS.
- `MAV_SMSClass *mav_SMSClass_HBB` – the Hierarchical Bounding Box SMS.

C.3.6 SMS callback functions

The following SMS callback functions are defined on initialisation to act on an SMS.

- `MAV_SMSCallback *mav_SMSCallback_delete` – a handle to the "delete SMS" callback.
- `MAV_SMSCallback *mav_SMSCallback_objectRmv` – a handle to the "remove object from SMS" callback.
- `MAV_SMSCallback *mav_SMSCallback_objectAdd` – a handle to the "add object to SMS" callback.
- `MAV_SMSCallback *mav_SMSCallback_intersect` – a handle to the "SMS-line intersection test" callback.
- `MAV_SMSCallback *mav_SMSCallback_pointerReset` – a handle to the "reset SMS contents pointer" callback.
- `MAV_SMSCallback *mav_SMSCallback_pointerPush` – a handle to the "push SMS contents pointer" callback.
- `MAV_SMSCallback *mav_SMSCallback_pointerPop` – a handle to the "pop SMS contents pointer" callback.
- `MAV_SMSCallback *mav_SMSCallback_objectNext` – a handle to the "get next object in SMS" callback.
- `MAV_SMSCallback *mav_SMSCallback_execFn` – a handle to the "execute function" callback.
- `MAV_SMSCallback *mav_SMSCallback_empty` – a handle to the "SMS empty" callback.

- `MAV_SMSCallback *mav_SMSCallback_size` – a handle to the “get SMS size” callback.
- `MAV_SMSCallback *mav_SMSCallback_objectContains` – a handle to the “SMS contains object test” callback.

Appendix D

Initialisation options

There are a number of ways of defining the various options that control the initialisation process. In order of lowest precedence first, they are:

- defaults values
- read from a config file
- application specified (using the `mav_opt_*` variables)
- via environment variables
- specified on the command line

The meaning of the various options, their default values, and how an application can define these is described in Appendix C (page 155). In the rest of this chapter we describe how the various options can be set from outside of the application.

D.1 Configuration file

Upon initialisation MAVERIK looks for a file called `.maverikrc` (or `maverik.ini` under Window and MacOS) first in the current directory and then in the user's home directory. If found this file is parsed. The format of the file is option-value pairs as follows:

```
verbose         (0|1)
fixedRnd        (0|1)
WMPlacement     (0|1)
singleBuf       (0|1)
multiSample     (0|1)
bindTextures    (0|1)
```

```
shareContexts   (0|1)
flush           (0|1)
finish          (0|1)
syncSwap        (0|1)
fullscreen      (0|1)
stereo          (0|quad|quad-sperate|two-wins)
restrictMouse   (0|1)
display         <X display string>
right_display   <X display string>
geometry        <X geometry string (widthxheight+xoff+yoff)>
right_geometry  <X geometry string>
name            <string>
right_name      <string>
splash          (0|1)
drawNormals     <length>
```

Multiple options should be separated by a newline. Option identifier is case insensitive.

D.2 Environment variables

The following environment variables can be used to control initialisation:

```
MAV_VERBOSE             (0|1)
MAV_FIXEDRND            (0|1)
MAV_WMPLACEMENT         (0|1)
MAV_SINGLEBUF           (0|1)
MAV_MULTISAMPLE         (0|1)
MAV_BINDTEXTURES        (0|1)
MAV_SHARECONTEXTS       (0|1)
MAV_FLUSH               (0|1)
MAV_FINISH              (0|1)
MAV_SYNCSWAP            (0|1)
MAV_FULLSCREEN          (0|1)
MAV_STEREO              (0|quad|quad-separate|two-wins)
MAV_RESTRICTMOUSE       (0|1)
MAV_DISPLAY             <X display string>
MAV_RIGHT_DISPLAY       <X display string>
MAV_GEOMETRY            <X geometry string>
MAV_RIGHT_GEOMETRY      <X geometry string>
MAV_NAME                <string>
MAV_RIGHT_NAME          <string>
MAV_SPLASH              (0|1)
MAV_DRAWNORMALS         <length>
```

D.3 Command line arguments

In order to use the following command line arguments to control initialisation, `mav_initialise` must be used to initialise MAVERIK:

```
-verbose
-silent
-[no]fixedRnd
-[no]WMPlacement
-[no]singleBuf
-[no]multiSample
-[no]bindTextures
-[no]shareContexts
-[no]flush
-[no]finish
-[no]syncSwap
-[no]fullscreen
-[no]stereo               (0|quad|quad-separate|two-wins)
-[no](restrictMouse|lockMouse)
-display                  <X display string>
-(geometry|geom)          <X geometry string>
-name                     <string (quoted if containing spaces)>
-(right_display|rdisplay) <X display string>
-(right_geometry|rgeom)   <X geometry string>
-(right_name|rname)       <string (quoted if containing spaces)>
-[no]splash
-drawNormals              <length>
-mavhelp
```

Appendix E

MAVERIK Frequently Asked Questions

```
     Frequently asked questions about GNU Maverik.

The latest version of this document can be found online at
http://aig.cs.man.ac.uk/maverik/faq.htm

Last changed: 4th June 01

-------------------------------------------------------------

Q1: Will Maverik run on <Insert your OS here>?
Q2: Does Maverik support the <Insert Peripheral Here>?
Q3: Does Maverik support the <Insert Graphics Card Here>?
Q4: My Maverik program doesn't seem to work.
Q5: I get a "error in loading shared libraries" or "rld: Fatal
    Error: Cannot Successfully Map soname" error.
Q6: I get a "MAV_HOME environment variable not set" error.
Q7: I get a "can't open avatar curve file walking.cset" error message
    when running the avatar example.
Q8: Why does Maverik run slowly on my old SGI?
Q9: Sometimes after my Maverik application has finished, the keyboard
    repeat is disabled.
Q10: Where are the pull-down menus, sliders and other widgets?
Q11: Why is there no collision detection?
Q12: Can read in objects defined in <Insert File Format Here>?
Q13: My objects appear to be clipped to planes that don't coincide
     with the view frustum - I'm getting objects disappearing when
     they get near the edge of the window.
Q14: Where is Spot The Dog?
Q15: Is Maverik thread-safe?
Q16: What about multi-distributed users?
Q17: Why are there 'get' functions, but no complimentary 'set' functions?
     For example, there is a mav_callbackGetMatrixExec but no equivalent
     mav_callbackSetMatrixExec?
Q18: I sometimes see a colour banding effect on objects - I get a
```

Q1: Will Maverik run on <Insert your OS here>?

Maverik is available as source code and should compile under Windows, MacOS and on UNIX systems - essentially any system that has OpenGL, Mesa (version 3.1 or above), IrisGL or DirectX (version 8). However, while it is possible to use any of these libraries, OpenGL/Mesa is currently the best supported library for Maverik to use.

Maverik is known to run on RedHat 5.2 and 6.x; FreeBSD 3.2; SuSE 7.1, Irix 5.3, 6.3 and 6.5; SunOS 5.7; MacOS and Windows 98, 2k and NT. This list is not intended to be exhaustive but simply reflects operating systems that we, or others, have access to and tried Maverik with. Ports to other UNIX platforms should be fairly trivial and we belive the code to work on Window 95.

Since we at the University of Manchester do not have access to SunOS, SuSE, FreeBSD, or MacOS; new releases of Maverik cannot be tested to ensure they will compile error-free on these platforms.

Feel free to contact us if you want more details of exactly what porting to other platforms would involve.

Q2: Does Maverik support the <Insert Peripheral Here>?

A standard compilation of Maverik provides supports for a desktop mouse, keyboard and screen. This makes it easy to try out the examples and demonstrations.

The configuration of 3D peripherals used in VR labs tends to be site specific. Code is included in the distribution to support Polhemus FASTRAK and ISOTRAK II six degree of freedom trackers (optionally coupled to Division 3D mice); Ascension Flock of birds (ERC only);

Spacetec SpaceBalls and SpaceOrb360s; Magellan Space Mouse; InterSense InterTrax 30 gyroscopic trackers; 5DT data gloves; and a serial Logitech Marble Mouse. With modification other similar specification 6 DOF input devices/tracking technology can be supported. This code is not compiled by default since it is not relevant to everyone and requires some manual configuration. See the README in the src/extras directory for more information.

We have also supported more peculiar peripherals in our own lab: IBM ViaVoice speech recognition system; Microsoft SideWinder Force-Feedback joystick and our homebuilt MIDI server. These are relatively uncommon devices and so are not included in a "standard" Maverik release. However, if your interested in this code drop us an e-mail.

Q3: Does Maverik support the <Insert Graphics Card Here>?

Maverik ultimately makes calls to a well supported graphics library (OpenGL, IrisGL or DirectX) to perform its rendering. Therefore, if these libraries are hardware accelerated by your graphics card, then Maverik will be accelerated.

For SGI machines this process is seamless. Unfortunately, for PC’s things get a little more complicated. Mesa, the freely available OpenGL work-a-like, supports hardware acceleration for a number of graphics boards [see http://mesa3d.sourceforge.net]. Alternatively, the graphic card vendor may supply drivers. We have verified Maverik with the following cards: Voodoo, Voodoo2, TNT, TNT2, GeForce 256 and GeForce2 based boards. See the Mesa web page for more information and the README.3DFX file in the Mesa distribution for how to compile Mesa to take advantage of Voodoo based graphics boards.

Maverik should automatically detect that a Voodoo based card is present and that hardware acceleration has been requested (it determines this from the MESA_GLX_FX environment variable). Under these circumstances, the default window size is changed to a Voodoo compatible 640x480 and the mouse pointer is restricted, by default, to stay within the graphics window.

Q4: My Maverik program doesn’t seem to work.

That’s not a question.

Q5: I get a "error in loading shared libraries" or "rld: Fatal
    Error: Cannot Successfully Map soname" error.

Maverik libraries are dynamically loaded and therefore you may need to

include their location in the dynamic library search path environment
variable LD_LIBRARY_PATH (or possibly LD_LIBRARYN32_PATH on
Irix6). E.g.

export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/local/maverik-6.1/lib

You may also need to include the location of the TDM libraries, if
used, in the LD_LIBRARY_PATH.

Q6: I get a "MAV_HOME environment variable not set" error.

Errr, like it says, your MAV_HOME environment variable is not
set. Some Maverik applications require this environment variable to
be set to point to the location of the Maverik distribution. E.g.

export MAV_HOME=/usr/local/maverik-6.1

Its good practice to set it regardless of whether your application
requires it or not.

Q7: I get a "can't open avatar curve file walking.cset" error message
    when running the avatar example.

Your MAV_HOME environment variable is incorrectly set.

Q8: Why does Maverik run slowly on my old SGI?

For older SGI machines (e.g. a Crimson) which are optimized for IrisGL
you need to run the Maverik setup script specifying the IrisGL option:
"setup --IrisGL" to generate a version of Maverik based on this
graphics library.

Note: IrisGL is no longer actively supported and its functionality may
differ from the OpenGL version of Maverik.

Q9: Sometimes after my Maverik application has finished, the keyboard
    repeat is disabled.

This problem has been fixed in version 5.4. Thanks to Miklos Szeredi
for providing the patch.

Q10: Where are the pull-down menus, sliders and other widgets?

Maverik is intended to be used to display a 3D "VR world" efficiently and flexibly. As such, providing GUI functionality for an application is outside of its scope.

That said, some GUI toolkits provide an OpenGL canvas widget which Maverik can use as its rendering window. GTK+ is such as a toolkit (support for other toolkits should be possible). Support for GTK+ must be specified to the setup script when compiling Maverik. See README-GTK for more information on GTK+ support. There is an example of using GTK+ in the examples/misc/GTK directory (only compiled if the GTK+ support is enabled).

Maverik has also been used with xforms. Here the Maverik rendering area and the GUI are separate windows - rather than Maverik being a sub-window forming part of a larger GUI window. There is an example of using xforms in the examples/misc/xforms directory (not compiled by default because it need the xform path specifying).

Q11: Why is there no collision detection?

At first it may appear strange that the standard Maverik navigation does not perform collision detection. What you should remember is that Maverik is a toolkit - it provides you with the functionality to easily detect if a collision has occurred, but does not dictate what happens as a result, that bit is upto you.

Its straightforward to implement navigation which performs collision detection to stop the users movement (an example of this is implemented in the programmers guide). The advantage of Maverik being a C toolkit is that it allows you to apply any additional constraints you wish. For example, collision with an object would only stop the user if:

(1) its volume was greater than some threshold.
(2) its of a certain shape or colour.
(3) its after 5pm on alternate Thursday’s.

Q12: Can read in objects defined in <Insert File Format Here>?

Maybe... There are a great number of modelers and file formats out there, but its not our job to support all of these. In fact, we support just three - VRML97 [http://www.vrml.org], Lightwave and AC3D [http://www.ac3d.org].

The VMRL97 parser uses the free CyberVRML97 for C++ library by Satoshi Konno (http://www.cyber.koganei.tokyo.jp/top/index.html). VRML97 support is *not* enabled by default - you must specify its use when installing Maverik. See INSTALL file for more details. Only the geometry of VRML97 files is read, no attempt is made to parse scripts,

URL’s, viewpoints etc... Furthermore, not all of the numerous ways in which the geometry can be defined are supported, e.g. concave polygons, colour-per-vertex.

AC3D is geometry modeler which, as well as creating and editing objects, can import them from a number of common 3D file formats (including 3DS, DXF, Lightwave and VRML1). Other packages, such as Crossroads and 3DC [there are links to these from the AC3D page], can translate between many different 3D file formats and the AC3D format. Thus, supporting AC3D provides a means of easily creating/editing objects and also indirect support for many common 3D file formats. Here’s the rub - a fully working version of AC3D cost $40 US.

Of course, there is nothing stopping you writting your own support for your favorite file format.

Q13: My objects appear to be clipped to planes that don’t coincide with the view frustum - I’m getting objects disappearing when they get near the edge of the window.

Your view direction or view up vectors probably aren’t normalised.

Q14: Where is Spot The Dog?

Usually under the table. Have a good look.

Q15: Is Maverik thread-safe?

Yes and no. The Maverik kernel is not currently re-entrant. It does not attempt to protect its own data structures from being damaged by multiple threads attempting to access them at once. So in this sense, it’s not theadable.

However, on GNU/Linux the system could be compiled using the _REENTRANT libraries and header files, so if you put your own protection around the Maverik calls, it would be safe to use them in threaded programs.

Q16: What about multi-distributed users?

Maverik was designed to provides the management of all the graphics and peripheral driving capabilities *for a single user* in a flexible, customizable and efficient manner.

A complementary system under development, Deva, provides a networked

multi-user environment on top of Maverik, with the ability to specify multiple active environments, laws etc.

Q17: Why are there 'get' functions, but no complimentary 'set' functions? For example, there is a mav_callbackGetMatrixExec but no equivalent mav_callbackSetMatrixExec?

Maverik doesn't hold it's own copies of data structures, so once you've 'got' a structure, that really is it. You can do whatever you like with that structure, and Maverik will just use that. There's no need to 'put it back' into Maverik.

Q18: I sometimes see a colour banding effect on objects - I get a saw-tooth pattern around the edges of objects?

These effects are probably due to limited depth buffer resolution and are particularly noticeable when using, but not limited to, Voodoo based cards and Mesa.

There are a limited number of bits comprises the depth buffer with the result that the depth values calculated for two objects whose distance from the eye is large but similar (or indeed the front and back faces of the same object if backface culling has not been enabled) can get quantized to the same value. The result of this is the object or faces may not be correctly depth buffered leading to one "poking through" the other. This "poking through" pattern moves around with the eye point as the depth values of the object/faces get quantized differently.

There are two ways in which you can reduce this effect:

1) Enable backface culling.
2) Reduce the depth range of your model thus allowing it to be more accurately represented with the limited number of bits available. This is achieved through the near and far clip distance set with the function mav_windowPersepectiveSet. It is more beneficial to increase the near clip plane distance that to reduce the far clip plane.

Q19: How are MAV_matrix's implemented/ordered?

Easy answer: its unimportant *provided* that you manipulate matrices via the functions Maverik provides and that you use the results of these operations as graphical transformations.

Complex answer:

OpenGL, Maverik and most graphics textbooks use a convention of

```
postmultiplying by column vectors, i.e. v' = M.v

Maverik stores the matrix, M, as a 4x4 array of floats. Maverik
follows the standard C convention and uses row-majored access to a 2
dimensional array, i.e. M[i][j] refers to row i column j. Thus, the X
translation term of a matrix, the top-right element, is accessed as
as M[0][3].

However, OpenGL expects to receive matrices as column-majored, and
therefore a 2 dimensional matrix implemented in C needs to be
transposed before being passed to OpenGL (see OpenGL programmers
guide). Maverik automatically performs this transpose.

------------------------------------------------------------

Q20: Are applications written for Maverik version 4.x compatible with
     version 5.x?

Possibly. Version 5 handles matrices in the manner described above,
i.e. row majored whereas version 4 used column majored. If you set
matrices via the functions Maverik provides, rather than accessing
individual elements, then this internal change should not prevent
back compatibility. If you do access individual elements, then you
will need to transpose the element indices to make a version 4
application work with version 5, i.e. m.mat[2][1] becomes
m.mat[1][2].

Also, version 5 fixed a bug in which roll and yaw angles were left
rather than right handed. So while the code will still compile and
execute, objects may no longer be oriented as before. Replacing
mav_matrixSet with mav_matrixSetOld will fix this, but it is
recommended that you correct the angles supplied to the
mav_matrixSet and mav_matrixRPYSet functions.

In version 4 the avatar's hands were specified relative to his body,
version 5 requires them in World coordinates.

------------------------------------------------------------

Q21: How do I recompile modified Maverik soure code, examples, or demos?

The top level Makefile in the Maverik distribution defines various
environment variables before traversing the sub-directories
performing a "make" in each. The Makefiles in the sub-directories rely
on these environment variables and so will not operate correctly if
they are directly executed.

So, if you modify the Maverik source code or examples then to
recompile them you must either:

(1) type "make" in the top level directory of the Maverik
    distribution, or
(2) Manually set the environment variables and type "make" in the
```

```
    sub-directory containing the modifications.

We suggest the first. Traversing the sub-directories where no changes
have been made is a fairly quick process.

The examples rely only on two environment variables, MAV_HOME and CC,
to indicate where Maverik is installed and what complier options to
use (this is documented in the MPG). The demos additionally rely on
the OPENGLINCL and OPENGLLIBS environment variables to indicate how to
include the OpenGL header files and how to link with the
libraries. The Makefiles for the Maverik source code rely on many more
environment variables and setting these by hand in not recommended.

-----------------------------------------------------------------

Q22: Why are some of the later chapters in the MPG missing?

Maverik is a large system and fully documenting it will take some
time. So far we have concentrated our efforts on the MPG and in
particular what a beginner to the system needs to know. Advanced
usage, such as adding support for new input devices or creating
your own types of SMS, has yet to be documented - its coming though.

-----------------------------------------------------------------

Q23: Why are some of the functional specifications poorly documented
     or blank?

See above. We have place holders for all of the functions and types in
the MFS, but simply haven't had the time to fully document, and
importantly, cross-link them all. As with the MPG, we have
concentrated on documenting the most common functions. The others are
being steadily added.

-----------------------------------------------------------------

Q24: Why do the man pages look horrible on SGI's?

Dunno. If I had to guess I'd say something like they use a different
version of groff than pod2man, which created the man pages, was
expecting. Live with it or use the HTML versions until this problem is
fixed.

-----------------------------------------------------------------

Q25: I get an "ld: cannot open -lGL: No such file or directory" error
     when compiling Maverik

Maverik 5.2 and before linked with -lMesaGL under Linux and
FreeBSD. Version 5.3 and later link with -lGL. This change coincides
with Mesa changing the name of the library it generates (libMesaGL.so
before version 3.1, simply libGL.so after). So, if Maverik can not
find libGL.so upgrade to version 3.1 of Mesa or soft link libGL.so to
```

```
be libMesaGL.so

------------------------------------------------------------

Q26: Why does mav_matrixRPYGet give the wrong results?

A. It doesn't, it give the correct answer - probably just not the one
you want :)

The conversion from an orientation matrix to a set of roll, pitch and
yaw values is inherently ill-defined. That is to say there are
multiple sets of RPY values which describe a given orientation - there
is no one-to-one mapping. For example, a RPY of (0, 0, 145) is
mathematically identical to one of (180, 180, 35) in that they both
give the same orientation.

mav_matrixRPYGet returns just one of the many possible RPY values
which can describe a given orientation. This may or may not be the
most "intuitive" set.

If you use all three RPY values together there should not be a
problem. What you cant do is modify one of the values in isolation and
expect to get sensible behaviour.

------------------------------------------------------------

Q27: Why are the numbers returned by mav_fps and mav_fps_avg incorrect?

A: mav_fps is based on the elapsed wall-clock time between the start
of mav_frameBegin and end of mav_frameEnd. (mav_fps_avg is simply
mav_fps averaged over a number of frames).

However, since an OpenGL implementation can buffer commands in several
different locations - including network buffers and the graphics
accelerator itself - the time recorded by mav_fps may not accurately
reflect the time it would take for the commands to complete.

In order to get an accurate time mav_frameEnd must wait until the
effects of all previously called OpenGL commands are completed. This
can be achieved by setting mav_opt_finish to MAV_TRUE.

Also, this buffering effect needs to be taken into account if you are
timing a sequence of graphics commands, for example in order to abort
rendering after a given elapsed time. The mav_gfxFinish command can be
called to flush the command buffers and wait until their effects have
been realised.

------------------------------------------------------------

Q28: What are the issues when dealing with semi-transparent objects?

A: In order to correctly deal with semi-transparent objects the
application must set the mav_opt_trans variable to MAV_TRUE before
```

initialising Maverik.

With this enabled a check is made before each object is rendered to determine if it is semi-transparent. If it is not, the object is rendered immediately; if it is semi-transparent then the object is not rendered but stored in a list for processing later.

At the end of the frame, after all opaque objects have been rendered, the list of semi-transparent objects is traversed. These objects are rendered in back-to-front order - that is the furthest object from the eye point is rendered first, the closest to the eye point last.

In order for Maverik to determine if an object is semi-transparent it executes the getSurfaceParams callback on it. Similarly, the BB callback is executed to obtain the object's bounding box (and hence position) in order to perform the depth sorting. A user-defined class of object would need to provide both of these callbacks if the object is to be correctly treated when semi-transparent.

Note that semi-transparent objects which overlap in space may not appear correctly since the depth sorting effectively treats each object as a point.

Backface culling should be enabled when using semi-transparent objects to avoid the "far-side" of the object being visible.

Q29: Are applications written for Maverik version 5.x compatible with version 6.x?

A: No, but the changes you need to make to a 5.x application in order for it to work with Maverik 6.x are quite small and mechanical:

1. Initialisation - mav_initialise in 5.x has been renamed to be mav_initialiseNoArgs in 6.x and mav_initialiseArgs in 5.x has been renamed to mav_initialise in 6.x.

2. Frame functions - The prototype of MAV_frameFn changed in 6.x to allow arbitrary data to be passed to the function. The easiest way to upgrade any such functions from 5.x to 6.x is to make them take an ignored void * parameter and to call the mav_frameNAdd/Rmv functions with a NULL argument. So,

```
  void fn(void)
  mav_frameFn0Add(fn);
```

in 5.x becomes:

```
  void fn(void *ignored)
  mav_frameFn0Add(fn, NULL);
```

in 6.x.

```
3. TDM - TDM libraries are now specified at run time and dynamically
loaded rather than being statically linked into Maverik. See
examples/misc/TDM/tdm.c for an example of how the library is
specified.
```

Concepts index

Functions index

This index lists the MAVERIK functions (and a few OpenGL functions) mentioned in this manual. For the full list of all MAVERIK functions, please refer to the MAVERIK Functional Specification.

Types, Variables and Constants index

This index lists the MAVERIK types, variables and constants mentioned in this manual. For the full list, please refer to the MAVERIK Functional Specification. We use the following typographical conventions: **types** begin `MAV_` and are then in mixed case (example: MAV_viewParams); **variables** begin `mav_` and are then in mixed case (example: mav_frameCount); and **constants** are all in uppercase (example: MAV_ID_MATRIX).

Bibliography

[1] Andy Colebourne. AC3D Modeller. http://www.ac3d.org.

[2] J. Cook and S. Pettifer. Placeworld: An integration of shared virtual worlds. *Accepted for publication in SIGGRAPH 2001 Sketches and Applications Program*, August 2001.

[3] Jon Cook, Toby Howard, Roger Hubbold, Martin Keates, Simon Gibson, Alan Murta, Steve Pettifer, and Adrian West. MAVERIK *Functional Specification*. Advanced Interfaces Group, Department of Computer Science, University of Manchester, June 2001.

[4] Jon Cook, Roger Hubbold, and Martin Keates. Virtual Reality for Large-Scale Industrial Applications. *Future Generation Computer Systems*, 14:157–166, 1998.

[5] M. Glencross, T. Howard, and S. Pettifer. Iota: An approach to physically-based modelling in virtual environments. In *Proc. IEEE VR2001*, March 2001.

[6] M. Glencross and Alan Murta. A virtual Jacob's ladder. In *Proc. GraphiCon-99*, August 1999.

[7] Mashhuda Glencross and Alan Murta. Managing the complexity of physically based modelling in virtual reality. In *Fouth international conference of Computer Graphics and Artificial Intelligence*, pages 41–48, Limoges, May 2000.

[8] Advanced Interfaces Group. http://aig.cs.man.ac.uk/deva/index.htm.

[9] T.L.J. Howard, S. Gibson, and A. Murta. Virtual environments for scene of crime reconstruction and analysis. In *Proc. SPIE/IS&T, San Jose, CA*, volume 3960, January 2000.

[10] R.J. Hubbold and N.P. McPhater. The use of virtual reality for training process plant operatives. In *Proc. CG Expo '94 Conference*, London, November 1994. Computer Graphics Suppliers' Association, Worcester, England.

[11] Roger Hubbold, Jon Cook, Martin Keates, Simon Gibson, Toby Howard, Alan Murta, Adrian West, and Steve Pettifer. GNU/MAVERIK: A micro-kernel for large-scale virtual environments. In *Proc. VRST'99, ACM Symposium on Virtual Reality Software and Technology 1999*, pages 66–73, London, December 1999. ISBN 1-58113-141-0.

[12] Roger J. Hubbold and Martin Keates. Landmarking for navigation of large models. *Computer and Graphics*, 23(5), October 1999.

[13] Roger J. Hubbold and Martin Keates. Real-time simulation of a stretcher evacuation in a large-scale virtual environment. *Computer Graphics Forum*, 19(2), June 2000.

[14] C. D. Murray, J. M. Bowers, A. J. West, S. R. Pettifer, and S. Gibson. Navigation, wayfinding and place experience within a virtual city. *Presence: Teleoperators and Virtual Environments*, 9(5):435–447, 2000.

[15] A.D. Murta, S. Gibson, T.L.J. Howard, R.J. Hubbold, and A.J. West. Modelling and rendering for scene of crime reconstruction: A case study. In *Eurographics UK Conference*, pages 169–173, Leeds, March 1998. Eurographics UK.

[16] S. Pettifer, J. Cook, and J. Mariani. Q-space: a virtual environment for interactive abstaract data visualisation. In *Proc. VRIC 2001*, May 2001.

[17] S. Pettifer, J. Cook, J. Mariani, and J. Trevor. Exploring realtime visualization of large abstract data spaces with QSPACE. In *Proc. IEEE VR2001*, March 2001.

[18] S. Pettifer, J. Cook, J. Marsh, and A. West. Deva3: Architecture for a large scale virtual reality system. In *Proc. ACM Symposium in Virtual Reality Software and Technology 2000 (VRST'00)*, October 2000.

[19] S. Pettifer, J. Cook, A.J. West, C. Murray, J. Trevor, J.A. Mariani, A. Colebourne, and A. Crabtree. Exploring electronic landscapes: technology and methodology. In *Proc. SPIE/IS&T, San Jose, CA*, volume 3960, January 2000.

[20] S. Pettifer, S. Gibson, and A. West. Visualising the virtual cityscape. In J. Mariani J, M. Rounce-field, J. O'Brien, and T. Rodden, editors, *Visualisation of Structure and Population with Electronic Landscapes*. Lancaster University Press, 1998.

[21] S. Pettifer and A. West. Deva: A coherent operating environment for large scale virtual reality applications. In *Proc. Virtual Reality Universe 1997*, April 1997.

[22] S. Pettifer and A. West. Worlds within worlds, on the topology of virtual space. In *Proc. Cyberconf 97*, May 1997.

[23] D. Smith, S. Pettifer, and A. West. Crowd control: lightweight actors for populating virtual landscapes. In *Proc. Eurographics UK 2000*, pages 65–71, April 2000.

[24] Dongbo Xiao and Roger Hubbold. Navigation guided by artificial force fields. In Joelle Coutaz and John Karat, editors, *Proceedings CHI '98 Technical Programme*, pages 179–186. ACM SIGCHI, ACM, April 1998.