GNU/MAVERIK
A micro-kernel for large-scale virtual environments

Roger Hubbold, Jon Cook, Martin Keates, Simon Gibson, Toby Howard, Alan Murta, Adrian West, Steve Pettifer
Advanced Interfaces Group, Department of Computer Science,
University of Manchester, Oxford Road, Manchester M13 9PL, UK
http://aig.cs.man.ac.uk/

Abstract

This paper describes a publicly available virtual reality (VR) system, GNU/MAVERIK, which forms one component of a complete 'VR operating system'. We give an overview of the architecture of MAVERIK, and show how it is designed to use application data in an intelligent way, via a simple, yet powerful, callback mechanism which supports an object-oriented framework of classes, objects and methods. Examples are given which illustrate different uses of the system, and typical performance levels.

CR categories and subject headings: I.3.7 Three-Dimensional Graphics and Realism (Virtual reality), I.3.2 – Graphics Systems, I.3.3 – Picture/Image Generation, I.3.4 – Graphics Utilities (graphics packages, software support), I.3.5 Computational Geometry and Object Modeling (Modeling packages, Object hierarchies), I.3.6 Methodology and Techniques (Graphics data structures and data types)

1 Introduction

This paper describes a publicly available virtual reality (VR) system, called MAVERIK, which has been under development for four years. In February 1999 MAVERIK was first released as free software under a GNU General Public License, and has been adopted by the Free Software Foundation as an approved component of the GNU project. Three previous papers described a prototype version of the system and some early applications [14, 5, 6]. In this paper we describe the architecture and design features of the released version of MAVERIK.

MAVERIK forms one component of a complete 'VR operating system', the other component of which is called Deva [19]. The latter provides a higher-level operating environment supporting multiple users, distributed shared environments, and multiple persistent concurrent environments. Within this wider framework, MAVERIK is designed to support high-performance rendering, including large-model processing, customised representations of VEs for different applications, and customisable techniques for interaction and navigation. Although it is a component of the larger system, MAVERIK can also be used stand-alone to build complete VR applications for individual users. In a multi-user environment, MAVERIK supports each user's view of, and interaction with, a virtual world.

We refer to MAVERIK as a *micro-kernel*, a term borrowed from operating systems terminology. It provides a framework and a set of core functions for constructing VEs, but attempts to do this in a way which does not force applications to use inappropriate structures for representing VEs. In this regard it has been strongly influenced by previous experience of designing and using high-performance computer graphics systems. Fundamentally, MAVERIK is concerned with how to represent diverse environments efficiently, how to manage large models, and how to customise behaviours and interaction. In the remainder of this paper we outline the design of the kernel itself and of a set of supporting modules which provide the functionality necessary to construct and interact with VEs. We also illustrate use of the system, and give an indication of its performance.

2 Virtual environment representations

One defining characteristic of a VR system is the way in which representations of a virtual environment are stored and manipulated internally. MAVERIK does not rely on one single representation. This has had a fundamental influence on its architecture and is one of the most important ways in which it differs from many other VR systems.

The issue of how best to represent a virtual environment (VE) inside a VR system is problematic. It is not a major concern if the required VE has limited possibilities for interaction, such as a simple walk-through, or interaction with only a small number of objects. But, as the complexity of an environment grows, and it is required to associate appropriate behaviours and affordances with objects, then it becomes essential that the internal representation can be readily tailored to suit the particular type of environment being modelled. These modelling requirements may vary widely across applications: for example, the representations needed by a design project in the motor industry will be quite different from those for an information system for the citizen.

2.1 Modelling in computer graphics and VR systems

A majority of VR systems adopt the approach of standardising on a particular internal representation of an environment. Examples include DIVE [3], dVS [9], NPSNET [24], Superscape and WorldToolKit. The advantage of a pre-defined representation is that a large number of 'standard' capabilities can be provided as part of the system's design and implementation. However, choosing a common internal representation, suitable for widely differing applications, is a difficult task, inevitably involving a trade-off between conflicting interests.

In both computer graphics and virtual reality systems we can distinguish between two kinds of representation. The first – which is a key driver for future virtual environments – is the modelling of the environment itself, including the behaviour of the environment and the objects within it. This type of modelling will vary widely between different applications, but must be closely matched to individual environments in order to give them meaningful semantic rules and behaviour.

The second type of modelling is that required for efficient display. Arguably, this has traditionally dominated the design of computer graphics systems, and its strong influence on VR systems can be observed. Published representations include the PHIGS structure store [11], and the Inventor, Performer [20] and VRML [18] scene graphs, and more recently the Java3D Toolkit [21]. In essence, structure stores and scene graphs are quite similar: both use a hierarchical representation to model objects, with different parts of the hierarchy linked using instances and affine transformations. Such techniques have long been established as a convenient way to describe the *graphical* attributes of scenes, but they may bear little relation to the physical characteristics of objects and environments. Consequently, these other aspects must be modelled separately, and this results in a significant duplication of code and data – an application must mirror the functionality of the rendering software in order to correctly simulate behaviours [12]. In effect, the application program ends up maintaining two separate representations – one for modelling *behaviours*, and one for *rendering*. It must also keep both representations synchronised. The time required to *build* a separate graphical representation may also be large: in a study of PHIGS, Hitschfeld *et al* found that the time needed to build a hierarchical representation could be as much as two orders of magnitude greater than the traversal and display time [10].

One reason why representations such as scene graphs are problematic is that a *data structure* is used to describe a scene, when a *language* would be a better choice. A language gives much greater freedom of expression: conditional clauses permit alternative display representations, and features such as geometry can be parameterised readily. A language also admits a wide range of application-specific representation schemes. For example, an application where this flexibility is useful is the design of pipe-work in a power plant, where an appropriate representation is a set of connections in the form of a graph with common nodes, rather than a strict hierarchy. Data structures like a scene graph cannot easily represent such a design. In a scene graph, geometric primitives are stored entirely within individual nodes, and cannot span different nodes. This means they cannot conveniently handle parameterised geometry – known as variational geometry in geometric modelling – other than those parameters which can be represented by simple affine transformations. Combining VRML with Java goes some way to addressing the issues of programmability, but still relies on the scene graph concept, and is subject to its limitations [2].

2.2 Immediate-mode rendering

An alternative to storing graphical data in a separate data structure such as a scene graph 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, C++, or Java. Calls to functions in a graphics library, embedded within the program, send data directly to the graphics hardware for immediate rendering. Using a language to generate output in this way addresses shortcomings of the scene graph, by supporting conditional display, parameterised descriptions, and arbitrarily structured data. SGI's OpenGL and Microsoft's Direct3D are examples of libraries which support immediate-mode rendering.

MAVERIK is designed for immediate-mode rendering using functions tailored for individual applications, rather than a standardised internal representation. With immediate-mode rendering, instead of maintaining two separate models – one for application data and one for graphical data – a *single representation* is employed. Data representation is customised to suit the application, and this often yields large savings in memory.

Figure 1: Example of a CAD model

Figure 1 shows part of an off-shore gas platform, generated by MAVERIK. It is a complex model, with a high level of detail. Many VR systems would store the graphical representation of this model using polygon meshes. Multi-resolution meshes, created by polygon simplification, are absolutely essential for models of this complexity in order to reduce rendering times. However, this further increases the storage requirements. For example, the model in Figure 1, if represented as polygons with three levels of detail, requires 500 megabytes of memory for the graphical data alone.

Handling models of this size, stored as polygons, is therefore not possible on a typical workstation or PC. In the UNC walk-through project, Aliaga *et al* [1] have used textured depth meshes, level-of-detail techniques, and occlusion culling to reduce visible polygons in a similarly complex model (a power plant containing 15 million polygons). They report that to get optimal performance requires 60 megabytes for model geometry and a further 80 megabytes for textured depth meshes. Their preprocessing times amount to 525 hours, and their final walk-through runs on an Onyx computer.

In contrast, when the model shown in Figure 1 is represented by parameterised primitives, such as cylinders, boxes, and sections of tori, the memory requirement reduces from 500 to 25 megabytes. These primitives can then be displayed by generating the polygons procedurally, using immediate-mode rendering. The functions which generate the polygons dynamically adjust the level of detail, using the usual metric of projected image size, and also apply context-sensitive processing to reject or simplify objects which are not of interest to the user for a particular task. One example of this is to treat key parts of a model as landmarks during navigation [13]. Context-sensitive culling of this type can be difficult to implement when a separate, standardised graphical representation is chosen, because much of the application-specific knowledge is discarded during model conversion.

Figure 2 illustrates a second example in which customising the representation permits efficient exploitation of application-specific knowledge. The characteristics of this model are quite different from the gas platform. In this case the virtual environment is a de-

tailed model of our Computer Science building at the University of Manchester. It is subdivided into rooms and a *cell and portal* data structure is used to accelerate rendering [17]. Several techniques are used to control level-of-detail: textures and light maps, and dynamic traversal of the hierarchy of patches resulting from the radiosity solution, so that greater detail is displayed when surfaces are close to the viewer. The cell and portal spatial data structure allows real-time walk-through on relatively modest computers, such as an SGI O2. Such a structure would be completely inappropriate for the gas platform, because of its open structure. Both of these examples were programmed with MAVERIK.

Figure 2: Radiosity model of the Computer Building. The white boxes show two portals to neighbouring rooms, used to cull the geometry.

3 MAVERIK

A consequence of our decision to allow both data representations and algorithms to be customised for individual applications is that the data describing an environment is assumed to exist *outside* the MAVERIK micro-kernel. For the kernel to provide useful functions, it must therefore define a framework for accessing this data, displaying it, and enabling the user to interact with the environment. MAVERIK supports object definition and management, large model display and management, interaction, navigation, collision detection and avoidance, plus all of the usual low-level features such as managing different display devices (e.g. stereoscopic projection displays, head-mounted displays), and handling input devices such as 3D trackers and voice recognition hardware.

The micro-kernel has an object-oriented structure. It defines a set of *classes* for different kinds of objects and data, and mechanisms for defining new classes. Customisation for different applications is achieved by defining *methods* associated with each class. It 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, including our undergraduate students. Methods are implemented by means of *call-back functions*, with data passed via generic pointer parameters. This call-back method has the disadvantage that it does not support inheritance, but with well-structured code this has not proved to be a problem. It may also appear clumsy compared with C++, but it has the advantage that methods can be altered dynamically, at run-time, by re-plugging pointer values, rather than relying on a compile-time binding of methods and classes. This feature is used by Deva to dynamically load different methods.

We know of three other systems which address some of the same issues as MAVERIK. Watsen and Zyda's Bamboo [22] supports configuration via dynamically loadable modules based on call-backs. So far as we can discover, Bamboo is in a less developed state than MAVERIK. However, unlike MAVERIK, Bamboo deals with shared VEs; in our work this aspect is handled by Deva [19], which is beyond the scope of this paper. Green's MRObjects [8] is another multi-user system. It implements a number of standard classes of objects using C++. New classes can be defined by extending the base class, using normal inheritance mechanisms. MRObjects adopts the accepted graphics practice of hierarchical modelling and provides built-in mechanisms for object representations. It is not distributed as source code, so it is difficult to judge how easy extending the system would be in practice. Kessler, Kooper and Hodges's Simple Virtual Environment (SVE) Toolkit [16] is in many respects the most similar to MAVERIK. It too uses call-backs to permit non-standard objects to be rendered via an underlying graphics library (GL or OpenGL plus X).

3.1 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 – a ladder, or a valve, for example.

The way to define different kinds of objects is to create a *class* for each one. This is effected 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 call-backs. Methods govern operations such as displaying primitives, computing their bounding boxes, or finding objects which are spatially closest to a given point. 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. 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 call-back functions every time a new application is implemented, MAVERIK provides *default methods* for common primitives such as polygons, polygon meshes, spheres, cylinders, cones, 'caps', tori, boxes, and sub-parts of these (e.g. 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 the kernel 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 the kernel can find the class of any registered object – and hence any associated methods – and can also pass to the call-back functions the generic pointer to the application data. Call-back functions perform a cast into a pointer of the correct type for the data.

This approach has three advantages. First, none of the application data is imported into, or replicated within, the micro-kernel, so the need to synchronise changes to multiple representations is avoided. Second, MAVERIK objects encapsulate all the information needed by the kernel to access data and methods stored exter-

nally within the application, so that object classes can be reused easily in other applications. Third, the method 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 serious legacy problem with large-scale databases and existing code.

3.2 Spatial management structures

Another defining characteristic 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 [15]. However, as with object storage, it is possible to find optimisations which capitalise on application-specific features to yield superior performance.

The MAVERIK micro-kernel provides a framework which permits customisation of spatial management methods. In a manner analogous to object definition, the kernel uses classes and methods to store and access spatial data. An application defines a class for each object storage technique, registers the call-back 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.

No assumption is made about how SMSs are stored, but default methods are supplied which implement a range of useful techniques. For example, one default class of SMS stores objects as a simple linked list, and processes them (e.g. displays them) in the order in which they were inserted. Another implements a hierarchy of bounding volumes, while a third supports a cell and portal structure, with a hierarchy inside each cell. Methods associated with SMSs include *object insertion*, *object deletion*, returning the *bounding volume* of an object, and *intersection* testing. However, as with objects, an application can define whatever new classes and associated methods are most appropriate.

3.2.1 Multiple SMSs

Although SMSs, as their name implies, are generally used for spatial management, they provide a general mechanism for storing object references within MAVERIK. How this is done, and the purpose for which they are used subsequently can be customised by an application by defining appropriate methods.

Objects can be inserted in any number of SMSs 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 main HBV structure are not needed during dynamic changes to the model. Because MAVERIK can manage multiple SMS structures simultaneously, the programming effort required to manage this is small.

A second example of multiple SMSs is to optimise the order in which objects are displayed. A first SMS is employed 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. Such optimisations can also be used for displaying stereoscopic views.

Data consistency is maintained because all SMSs store *references* to MAVERIK objects, which in turn contain references to the application-specific objects. The kernel 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.

3.3 The application interface

Figure 3 illustrates how the kernel uses externally supplied functions and data to customise its behaviour. In this figure, the central region represents the kernel itself; this maintains the data structures needed to provide the framework of classes and methods for management of objects. In this example, there are four objects. The first three are of type `'OBJ class 1'`, while the fourth belongs to `'OBJ class 2'`. The object classes contain methods for drawing (`Draw`), computing a bounding volume (`Bounding_box`) and testing for ray intersection (`Select`). The last of these is used for selecting objects interactively; application-specific methods can be added, as required. The methods are accessed as call-backs and are part of the application, not part of the kernel, as shown in the lower-left part of the figure. Instance variables (data structures) for objects are similarly accessed via a pointer and are shown in the upper-left corner. A similar object/class approach is used for spatial management structures, seen in the right part of the figure. Also shown are links between SMS data structures and object instances, and instances of the SMSs and the data about them. Although it may look complicated, the API which controls this is straightforward. The links have been shown in the figure to emphasise that the binding of methods to classes and SMSs, and the links between object instances and data, can be changed dynamically, and cheaply, at run-time.

Although not shown in this diagram, MAVERIK uses a similar call-back mechanism for registering event handlers and navigation functions. As mentioned previously, the complete system contains libraries of default methods for displaying and managing many common types of graphical primitive, for different forms of spatial management, and for navigation. These can be customised by adding extra data and code, or simply replaced by alternative versions which are intimately bound to the data structures used by the application.

It might be argued that other systems can be tailored in much the same way. However, in most systems a standardised internal representation, in the form of a polygon-based object store, or a scene graph, is employed. Changing this to use alternative representations can be quite complicated, requiring an intimate knowledge of the internal workings of the system, the alteration of code, and the re-compilation of the system. DIVE [3] is one example of a well-known system which operates in this way.

In MAVERIK it was a deliberate design feature that the actual object data is stored outside the kernel. Call-backs can be switched (re-registered) to change the dynamic behaviour of the system. Deva relies on this mechanism to change the way a user interacts with different environments as he or she moves between them. 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. We call this 'in-out' rendering: objects can present themselves in different ways depending on the observer's (or even the application's) context. More generally, entire environments can dynamically assume different representations and behaviours – simple ones viewed from 'outside' and complex ones from the 'inside'.

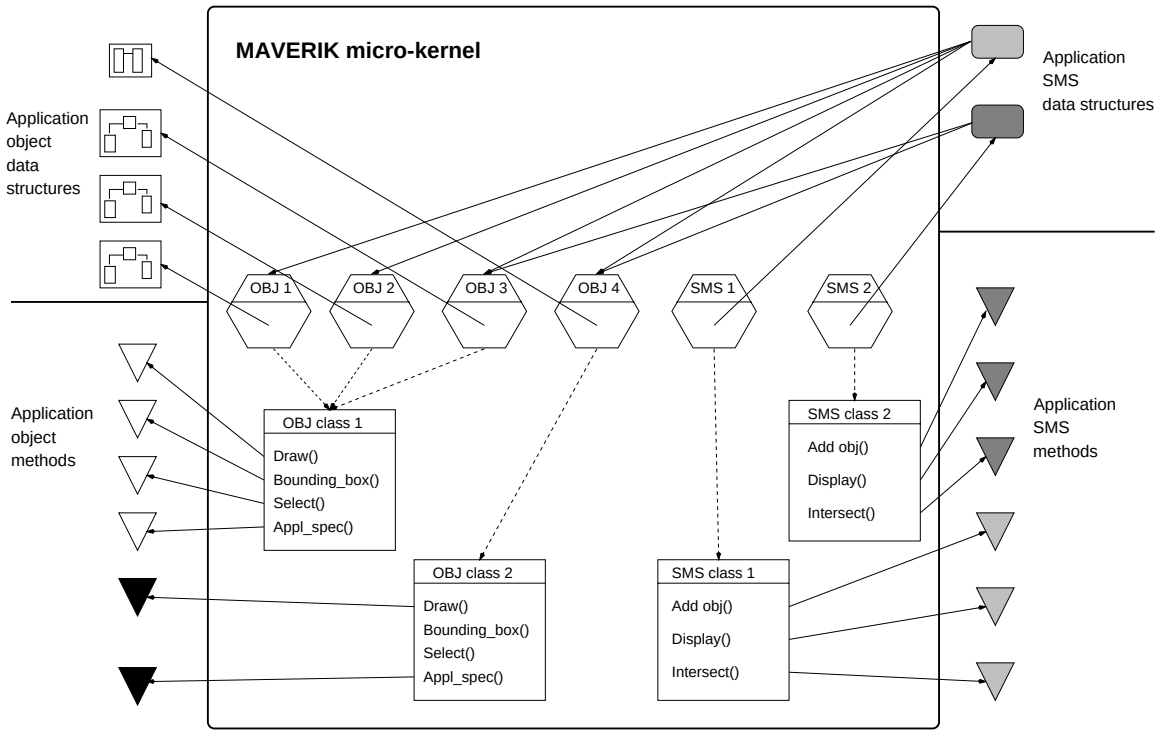

Figure 3: Schematic view of the MAVERIK application interface. The kernel stores information about classes of objects, SMSs, and any other application-defined classes (not shown here). Application data for each registered object, and methods (call-backs) for each defined class, are stored outside the kernel, and are referenced through generic pointers. Default classes and methods provide for rapid prototyping.

3.4 Kernel functionality

In addition to the application interface described above for object and spatial management definition, the kernel also provides the following functionality:

- **Input devices.** The kernel provides a mechanism by which input devices – mice, keyboards, 6 degree of freedom trackers, speech recognition, etc – can be supported. This is achieved by the application – or more commonly one of the supporting modules (see below) – defining up to three call-back functions per input device.

  The first of these polls the device to obtain its position in its local coordinate system: pixels relative to the window's corner in the case of a mouse, offsets in inches from the transmitter for a tracking system.

  The second call-back function maps the local coordinate system of the device into the world coordinate frame. For example, a mouse position is typically mapped onto the near-clip plane, and XYZ offsets from the transmitter are mapped as displacements from the eye point along the view-right, view-up and view-direction vectors.

  The third call-back function checks for and acts upon events generated by the device, typically by executing an application-defined call-back function specific to the class of object pointed to by the device.

  Applications are only required to define those call-backs appropriate to the input device: all three would be needed to support a mouse, but only the third for a keyboard or speech recognition system.

- **Graphical output.** An abstracted graphical output and window manager interface is defined by the kernel, so that applications can be written for different platforms without the need to address these system specific issues. In practice this is little more than a wrapper to OpenGL and X11. Alternate versions have been written for Iris GL and X11 functions, Windows, and Mac-OS, the last by a user at another site. There is even a version of the kernel which resolves calls to the interface without providing any graphical output; the Deva system uses a number of MAVERIK's functions in this way.

- **Math functions.** The kernel contains functions to perform common VR mathematical operations: conversion between Euler angles, 4x4 transformation matrices and quaternions; vector addition, subtraction, multiplication, normalisation, cross and dot products; transformation matrix inversion, transposition, multiplication and interpolation.

3.5 Supporting modules

It should be apparent that the micro-kernel itself is minimalist by design. Most of the hard work is delegated to the methods which support different classes, so that writing a VR application starting with only the micro-kernel itself would be a major undertaking. MAVERIK addresses this by providing a set of libraries, containing default methods and algorithms, known as the *supporting modules*.

These modules are distributed as source code and can be can be customised by adding extra data and code, or simply replaced by alternative versions which are intimately bound to the data structures used by the application.

Currently, there are six supporting modules, which are described below.

3.5.1 Common methods module

This defines methods which are almost always required: rendering, calculating an axis aligned bounding box, and calculating the intersection between an object and a line (used to implement picking when pointing at objects). It also contains common geometric methods such as intersecting a line with a plane, and re-aligning a bounding volume defined in a local coordinate frame to the global coordinate system.

3.5.2 Common objects module

This defines default data structures, object classes and methods for a range of 3D primitive shapes. These are: box, pyramid, cylinder, cone, sphere, half-sphere, circular torus, rectangular torus, polygon, polygon group, facet group, ellipse, half-ellipse, rectangle, polyline, text, teapot and composite object. Local transformations can be associated with these if desired, so that arbitrary orientations and scalings can be applied.

These primitives are stored in a parametric form, and the default display methods generate a graphical representation 'on-the-fly' using immediate-mode rendering. Level of detail processing, calculated from each object's projected size on-screen, is optionally performed to reduce the graphical load. Factors such as the minimum and maximum number of facets to be used, and the rate at which the number of facets used is reduced as the object recedes from the eye point, are under program control, and can be dynamically adjusted. In cases where it is more efficient to use a display list to render objects then appropriate display methods can straightforwardly replace the defaults.

The *composite object* type offers a convenient way to import objects from other systems. At present, such object definitions can be in VRML97 or AC3D format. AC3D is an editor implemented by the University of Lancaster [4], and it can import a range of other formats including 3D Studio, DXF, Lightwave, and VRML; any software which exports VRML could be used instead.

The abstracted graphical interface would be familiar to anyone with a working knowledge of OpenGL, so it is relatively easy to write new display methods and to tailor them for specific uses. For example, one of our users has implemented a NURBS class in order to model, display, and animate fish swimming.

The default structures in which object parameters are stored are referenced by pointer – no assumptions are made about how they are linked together or to other data. Thus, primitives can be grouped to form composite objects, and these can be stored in any suitable form, such as an array, a list, or a tree, embedded within the application's data structures.

3.5.3 SMS module

This module defines SMS methods to add and remove objects, intersect the objects with a line, and to cull to a volume of space passing the resulting objects onto a further function.

Three default SMS classes, their data structures and functions for the SMS methods, are also defined by this module. The first is a linked-list of objects (effectively unmanaged). The second is similar, but maintains the list in insertion-order. The third uses the axis-aligned bounding box call-back (see above) to construct a hierarchical bounding volume (HBV) data structure. Any new class of object, provided it defines a bounding box call-back, can therefore be used with this SMS.

Wrapper functions are provided which cull the structures to the view frustum and call the rendering method on the resulting objects.

3.5.4 Window module

The window module provides support for the mouse and keyboard as input devices, using the framework defined by the kernel. It defines additional call-back methods which an application can use to specify how objects respond to mouse and keyboard events.

The polling of the mouse position is achieved via a call to the abstracted graphical output, and window manager interface, provided by the kernel. A world coordinate position for the mouse is trivially calculated by mapping it onto the near clip plane.

Upon detecting a mouse button press or release, the callback function, which checks for and deals with events, constructs a vector from the eye point passing through the mouse's world coordinate position. Using this vector, the callback function for intersecting a line with an SMS (described above) is then executed for all SMSs to determine which is the closest intersected object. The call-back method for dealing with mouse events for this object is then executed if one has been defined by the application.

Additional mechanisms are provided for trapping events which occur when the mouse is pointing at an object irrespective of its type, and for events occurring when the mouse is not pointing at any object. A call-back function defined for the first of these cases normally takes precedence over the second case, which in turn takes precedence over a call-back function defined for the specific class of object.

The window module also provides support for stereo rendering by supplying routines to perform stereo offset calculations.

3.5.5 Navigation module

This module utilises the mouse and keyboard event call-back functions, defined above, to perform navigation.

For the case of mouse navigation, a function is defined which is executed wherever a mouse event occurs. This function arranges that while a mouse button remains depressed a function is executed at the start of each frame. This function measures the mouse position relative to where the event occurred, and calls two application-defined call-back functions, termed 'navigator functions', which arbitrarily transform the user view by a specified amount. One navigator function is executed with the specified amount of movement being equal to horizontal mouse displacement, the other with the vertical mouse displacement.

The module also defines default navigator functions to move along the major axis, along the view vectors, to roll, pitch and yaw the view, etc. An application defines which functions it wants to use for horizontal and vertical mouse movements, and a scaling factor to convert from pixels into the units used by the application. Customised navigator functions can be written by an application to exploit its semantics: for example, to restrict the user to walking on the pavement in the model of a cityscape (see later example).

The module also includes a generic force-field method of navigation, which supports ascending stairs and ladders, gravity, and dynamic collision detection. The force fields are repulsive forces which surround objects in an environment, and which gently assist a user in navigating around objects which they would otherwise become stuck against [23]. The method uses the SMS for rapid object testing, and the collision module (see next section).

3.5.6 Collision module

The collision module provides call-backs for computing intersections between different MAVERIK primitives. The approach uses a new hybrid algorithm, in which one object must be tessellated into polygons but the other uses the exact equations for the specific primitive. The polygonal object can be re-tessellated as required to achieve any desired degree of accuracy. These functions are employed by the force-field navigation method to test for collisions

during movement. They also support real-time collision detection during object manipulation.

4 Examples

Models such as that in Figure 1 have been tested on a variety of machines, ranging from a PC equipped with a Voodoo2 graphics card, an SGI O2, up to an Onyx/InfiniteReality system. We can demonstrate walk-through of models of this scale, running at frame rates of 11 frames per second on a 266 MHz Pentium II, with a Voodoo2 card [13]. To achieve this, application-specific culling techniques are used. This is only possible because MAVERIK can make direct use of knowledge about which objects are important for a given task. In fact, the algorithms used in this application are designed to smoothly vary the level of detail in order to maintain a chosen frame rate. Thus, when the same code is run on an Infinite-Reality machine, we can elect to have a much higher frame rate, or to keep the frame rate the same but see much more detail. In this application, a hierarchy of bounding volumes is used for spatial management.

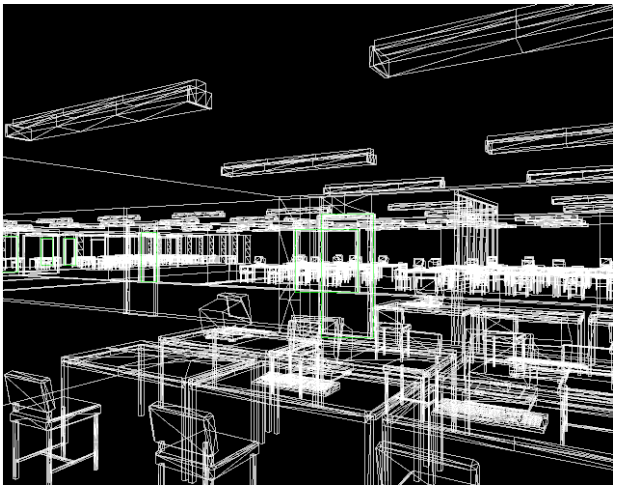

Figure 4: A wireframe view of the Computer Building. This is the same view as in Figure 2, but with no culling.

Figure 2 illustrates a walk-through of our building. A typical frame rate for this on the PC is 10 fps, although it varies between around 3 and 30 depending on the amount of furniture in view. The model contains quite a high-level of detail, with illumination computed using the VRAD perceptually-driven radiosity system [7]. In this case a cell and portal data structure is employed for the building, with a hierarchy of bounding volumes used inside each room. A wire-frame view of the scene without any culling is depicted in Figure 4.

A city walk-through provides a third illustration in Figures 5 and 6. Here, an application-specific occlusion culling method allows rapid rejection of buildings in the background, hidden by those near the viewpoint. The model is 40 city-blocks square, and it renders at a frame rate of 13 fps on the PC.

These examples illustrate different SMS techniques, which can be dynamically linked into MAVERIK. They demonstrate that quite complex models can be processed with modest resources by capitalising on tailoring the representations to the particular environment. Some of these models and programs are distributed free with the system. Further details can be found on our web pages, along

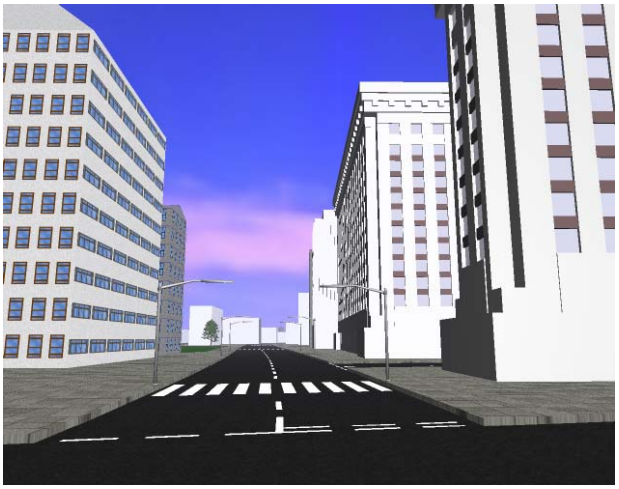

Figure 5: A view of a 40-blocks square cityscape.

with example images and MPEG movies captured in real-time on the PC.

5 Details of distribution

The MAVERIK system is now a component of the GNU project. It is available free under a GNU General Public License from http://aig.cs.man.ac.uk/ or http://www.gnu.org/software/maverik/maverik.html.

It is supported on SGI workstations using the OpenGL library, and on GNU/Linux PCs via the Mesa libraries. The distribution is available as compressed tar files, and Red Hat RPMs. The system contains more than 680 functions, although most applications can be implemented with a small number of these. The distribution includes source code, full documentation (PostScript, HTML, and man pages), demonstration programs for learning the system, and larger-scale demonstrations of how the system has been used.

6 Acknowledgements

This work was funded by the Engineering and Physical Sciences Research Council under grant number GR/K99701. We'd like to thank other colleagues and former and present postgraduate students in the Advanced Interfaces Group for their enthusiasm and contributions. We also thank our industrial partners: CADCentre Ltd, Brown & Root Ltd, Sharp Laboratories of Europe Ltd, and Conoco Ltd for the use of the CMS2 data.

References

[1] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerztlinger, E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. A framework for the real-time walkthrough of massive models. Technical Report TR 98-013, University of North Carolina at Chapel Hill, 1998.

[2] Don Brutzman. The Virtual Reality Modeling Language and Java. *Communications of the ACM*, 41(6):57–64, June 1998.

[3] Christer Carlsson and Olaf Hagsand. The MultiG Distributed Interactive Virtual Environment. In Lennart E. Fahlen and

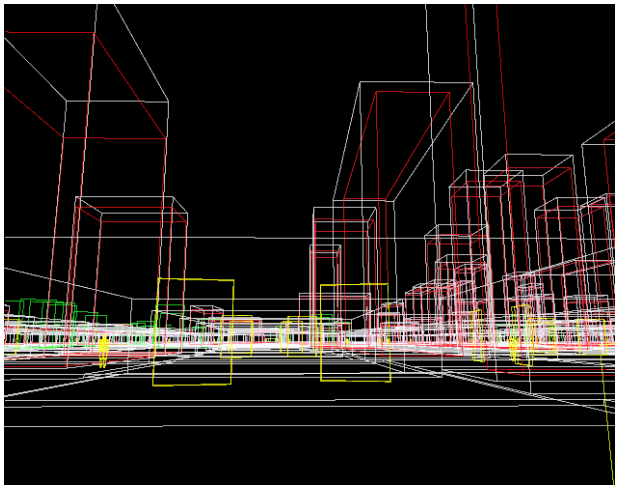

Figure 6: The same cityscape displayed in wireframe, without occlusion culling.

Kai-Mikael Jää-Aro, editors, *Proceedings of the 5th MultiG Workshop*, Swedish Institute of Computer Science, Box 1263, 164 28 Kista, Sweden, 1993.

[4] Andy Colebourne. AC3D Modeller. http://www.comp.lancs.ac.uk/computing/users/andy/ac3d.html.

[5] Jon Cook, Roger Hubbold, and Martin Keates. Virtual reality for large-scale industrial applications. In *Proc. EuroVR 97 Conference*, Amsterdam, November 1997.

[6] Jon Cook, Roger Hubbold, and Martin Keates. Virtual reality for large-scale industrial applications. *Future Generation Computing Systems*, 14(3/4):157–166, 1998.

[7] Simon Gibson. *Efficient radiosity simulation using perceptual metrics and parallel processing*. PhD thesis, Department of Computer Science, University of Manchester, September 1998. Available on-line as a University of Manchester Department of Computer Science Technical Report, UMCS-99-9-1.

[8] Mark Green. MRObjects. http://www.cs.ualberta.ca/ graphics/mrobjects/, 1999.

[9] Charles Grimsdale. dVS – Distributed Virtual environment System. Division Ltd, Bristol, UK.

[10] Nancy Hitschfeld, Dölf Aemmer, Peter Lamb, and Hanspeter Wacht. Performance evaluation of portable graphics software and hardware for scientific visualization. In M. Grave and W.T. Hewitt, editors, *Visualization in scientific computing*, number ISBN 3-540-56147-1 in Focus on Computer Graphics, pages 31–42. Springer-Verlag, 1994. Proc. 1st Eurographics Workshop on Scientific Visualization, Paris, April, 1990.

[11] T.L.J. Howard, W.T. Hewitt, R.J. Hubbold, and K.M. Wyrwas. *A Practical Introduction to PHIGS and PHIGS PLUS*. Addison Wesley, Wokingham, England, 1991. ISBN 0-201-41641-7, xv + 339pp.

[12] R.J. Hubbold and W.T. Hewitt. GKS3D and PHIGS – theory and practice. In W.T. Hewitt, M. Grave, and M. Roch, editors, *Advances in Computer Graphics IV*, chapter 3, pages 62–106. Springer-Verlag, 1991.

[13] Roger Hubbold and Martin Keates. Landmarking for navigation of large models. *Computers & Graphics*, 23(5), 1999. Special issue on Visibility – Techniques and Applications. In press.

[14] Roger Hubbold, Dongbo Xiao, and Simon Gibson. MAVERIK – The Manchester Virtual Environment Interface Kernel. In M. Göbel and J. David and P. Slavik and J.J. van Wijk, editor, *Virtual Environments and Scientific Visualization '96*, pages 11–20. Springer-Verlag/Wien, 1996. ISBN 3-211-82886-9.

[15] Timothy Kay and James Kajiya. Ray tracing complex scenes. *ACM Computer Graphics*, 20(4):269–278, 1986.

[16] Drew Kessler, Rob Kooper, and Larry Hodges. http://www.cc.gatech.edu/gvu/virtual/SVE/, 1997.

[17] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. *1995 Symposium on Interactive 3D Graphics*, pages 105–106, April 1995. ISBN 0-89791-736-7.

[18] David R. Nadeau. Tutorial: Building Virtual Worlds with VRML. *IEEE Computer Graphics & Applications*, 19(2), March – April 1999. ISSN 0272-1716.

[19] Stephen R. Pettifer. *An Operating Environment for Large Scale Virtual Reality*. PhD thesis, University of Manchester Department of Computer Science, January 1999. Available via: http://aig.cs.man.ac.uk/people/srp/research.html.

[20] John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real-time 3D graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[21] H. Sowrizal, K. Rushforth, and M. Deering. *The Java 3D API Specification*. Addison-Wesley, 1997.

[22] Kent Watsen and Mike Zyda. Bamboo – a portable system for dynamically extensible, real-time, networked virtual environments. In *Proc VRAIS'98*. IEEE Computer Society, 1998.

[23] Dongbo Xiao and Roger Hubbold. Navigation guided by artificial force fields. In *Proceedings of ACM CHI 98 Conference on Human Factors in Computing Systems*, volume 1, pages 179–186. ACM SIGCHI, Addison Wesley, April 1998. ISBN 0-201-30987-4.

[24] Michael J. Zyda, David R. Pratt, John S. Falby, Paul T. Barham, and Kristen M. Kelleher. NPSNET and the Naval Postgraduate School Graphics and Video Laboratory. *Presence*, 2(3):244–258, 1993.