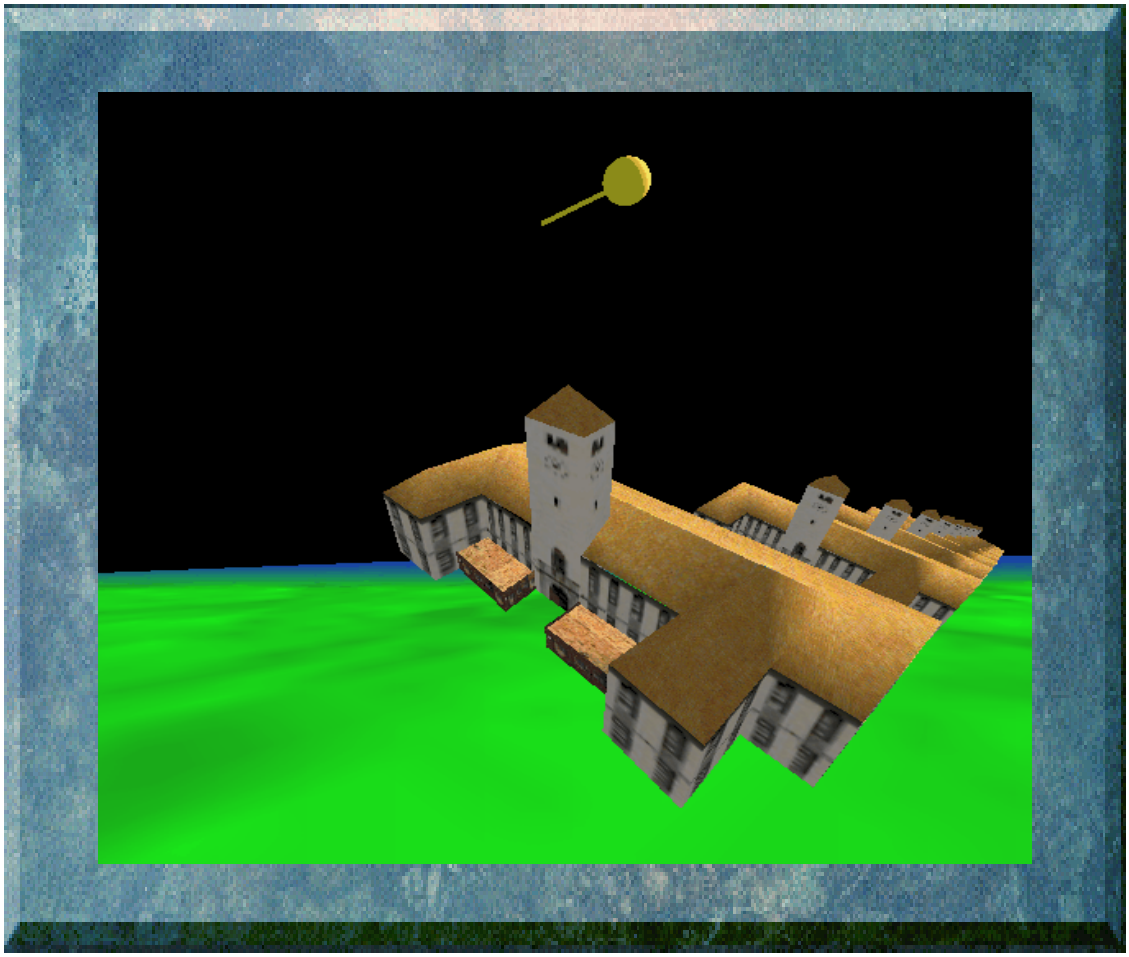
DEVELOPING EFFIECIENT GRAPHICS RENDERING COMPONENTS



Frederick WS Fourie

Supervisor: Prof Shaun Bangay

Abstract

This project investigates the development of a set of flexible, efficient and platform independent software components for rendering real-time virtual realty graphics. The object-oriented design allows specialised components to be easily developed using inheritance and polymorphism. Results of performance tests are presented and from this the following conclusions were drawn. Firstly, hardware graphics acceleration is crucial for successfully implementing interactive rendering; secondly, texture mapping is identified as a major target for future optimisation; lastly, the current feedback optimisation algorithm could be improved and suggestions are put forward as to how this could be achieved. Problems encountered are discussed and the thesis concludes with a discussion on future work and possible extensions.

Keywords: Interactive graphics, virtual reality, feedback optimisation, OpenGL, object-oriented programming

Contents

Figures and Graphs

1
Introduction

One does not have to look very far today to find the world of Virtual Reality. This technology, born out of the success of the information revolution, is enhancing the way people work and perceive the world in diverse fields, ranging from telemedicine and molecular engineering to creating cartoons and computer games. With this in mind, it was inevitable that sooner or later one might wonder as to what is involved in producing these pseudo-realistic images. And that is the main driving force behind this project: researching and implementing advanced rendering techniques with particular regard to real time virtual reality systems.

To make the work immediately relevant, I chose to implement my findings as a set of flexible object classes that could be used in the Rhodes University Centre Of Excellence (COE) CorGi project, the second generation Rhodes University Virtual Reality Environment. The first generation rendering software had proved to be too inefficient to provide real time feedback of large environments and so it became one of the primary aims of my project to provide efficient, real time rendering of a specified virtual environment. Other researchers could then use these classes to provide the rendering services required by their projects, where these projects could range from constructing entire three-dimensional virtual worlds to designing intuitive three-dimensional user interfaces.

It is important that the reader realises that this project was not aimed at developing some new and amazing rendering algorithm. That would require a substantial background in computer graphics. Rather, the aim was to first build a solid theoretical foundation and then use that foundation (in conjunction with existing graphic tools and standards) to implement the most efficient solution possible. Once this solution was in place, it's performance was carefully monitored and the results documented.

Fundamentally, this project is about developing a set of related rendering services. Therefore, it would be wise to first present the reader with an overview of the entire system before diving into the details. This should put each of the detailed sections that follow into perspective and provide the reader with insight as to how the various components of the system interact and rely upon one

another.

The major part of this report deals with describing the design and implementation of rendering services such as basic structural rendering, colour, smooth shading, blending and texture mapping. Each section will define the problems encountered in providing a particular service and the eventual solution to these problems. The reader will also find that each solution is aimed at providing the most optimal solution and each section contains a discussion on any optimisations used during the design and implementation of that particular service.

An important component of the project has been the development of a feedback mechanism that dynamically monitors and adjusts the rendering quality of a VR system to maintain a specified level of performance. This work goes beyond previous services offered by RhoVer and is assigned a chapter of its own.

Following the above sections, I present quantative results of performance tests on various platforms and discuss the results - both expected and unexpected. From these results, conclusions are drawn and the report ends with a discussion on possible extensions and future work.

2

System Overview

The set of component classes (hereafter referred to simply as components) that cooperate to provide real time virtual reality (RTVR) rendering services consists of eight components, each with unique tasks that contributes to the final result. A useful paradigm to keep in mind as we progress is that of a biological system: a collection of specialised organs (components) working together towards a common goal (real time virtual reality). A second paradigm that drove the analysis and design of the components is that of LEGO blocks: create a set of flexible components that are snapped together to build an application.

The entire system is built on top of the OpenGL® graphics library, the OpenGL utilities library and

the graphics library utility toolkit (glut), written by Mark Kilgard. However, to facilitate portability to another graphics library, the **VROutputDevice** component was created. It is the responsibility of this component to create a platform independent *rendering context* (the drawing window) as well as managing the three-dimensional rendering volume to which our application will draw.

The **VRSink** is responsible for drawing the VR scene to a rendering context(s). It does this by coordinating and controlling a VROutputDevice. The VRSink also instantiates the VRSinkMonitor and VRCapabilities objects, sets performance thresholds and enables feedback optimisation.

The **VRSinkMonitor** component, once enabled, will monitor the overall performance of the application and dynamically alter the complexity of the rendering process in an attempt to ensure that performance levels remain above a frame rate specified by the application programmer. This process will henceforth be referred to as *feedback optimisation*.

To accomplish feedback optimisation, the VRSinkMonitor instructs the **VRCapabilities** component to enable or disable various aspects of rendering such as textures, colours, smooth shading, etc. These aspects are referred as *capabilities* and it is the responsibility of the VRCapabilities component to keep track of which capabilities are currently enabled and which are not.

The **TextureManager** component creates and manages texture objects of all the texture maps used by the application. By having a separate class managing all textures as opposed to each object managing its own textures, it is possible to avoid identical textures being stored and processed more than once by the application. The TextureManager stores data relating to textures in a linked list of **TextureListNode** components.

Lastly, we come to the come to the component that represents and renders an object or entity in the virtual world: the **VisualRepresentation**. This component is responsible for inputting and parsing structural, colour and texture data, synthesizing this data into an accurate visual representation and then rendering this representation using the currently enabled capabilities. To support *collision detection*, the VisualRepresentation uses bounding spheres around each entity in the virtual world and this sphere is described by the data members of an associated **BoundingSphere** component.

3
Background

Now that the reader has some idea (albeit superficial) of what the system encompasses, it is time to start exploring in detail the problems that I encountered and their eventual solutions. Much research and thought went into finding the best possible solution to each problem and this section aims at providing the reader with the relevant background necessary to understand the following arguments.

3.1 Graphics libraries, standards and the ARB

A graphics library (such as OpenGL®, DirectX or Borland's Graph unit) can be described as a set of commands that interfaces to and controls the graphics hardware of a computer system. Various graphics libraries interface at different levels with the hardware, which plays an important factor in determining rendering speed. My first task involved choosing the correct graphics library (gl) and after looking at various nominees, the choice came down to two candidates: Microsoft's DirectX and OpenGL. I didn't have to look very far to realise that the OpenGL library was the obvious candidate for our work, for various reasons.

Firstly, and most importantly, the OpenGL library is a mature, open standard. This library was developed years ago by Silicon Graphics International (SGI), a company focussed solely on developing and producing high performance graphic systems. SGI has used OpenGL as the basis for sophisticated toolkits such as *Open Inventor*, and has encouraged the graphics industry to accept it as a standard gl by licensing the source for a minimal fee. In 1990, a consortium of companies formed the OpenGL Architecture Review Board, aimed at standardising the development and implementation of OpenGL libraries for different platforms (known as *bindings*). Any extensions to the open standard has to be approved by the ARB before being accepted as part of the next version[1]. Unlike OpenGL, DirectX is a proprietary gl developed and marketed solely by Microsoft. DirectX is also fairly new and updates are released regularly, a sign that DirectX is still undergoing major changes.

[1] The current version is 1.2, but few implementations conforming to the standard exist

It is also the opinion of many in the industry that DirectX is directed primarily at game developers and not well suited for large, complex visualisation applications. It's well known that DirectX provides better performance than OpenGL - for game platforms. On these systems, graphics acceleration is primarily achieved via software, unlike dedicated graphics workstations where OpenGL rules unchallenged. However, as graphics adapters progressively support more hardware acceleration, OpenGL is rapidly gaining in popularity amongst developers of various applications, including computer games. This is reflected on the ARB OpenGL homepage[1], where reports of new products using or supporting OpenGL are constantly being published. Furthermore, OpenGL includes features such as client-server support (for distributed rendering), platform independent code and support libraries as well as advanced mechanisms such as display lists and texture objects for optimising rendering performance. So, for those of us with more serious applications in mind, OpenGL provides clear advantages over DirectX.

A standard distribution of OpenGL includes 3 libraries:

1. gl
2. glu
3. glut

In order to keep the core library (gl) small, flexible and platform independent, OpenGL does not include commands for handling windows, obtaining user input or high level commands for describing three dimensional objects. The first two facilities can be implemented by using platform specific APIs or by using the graphics library utility toolkit (glut), which also provides commands for rendering basic three dimensional objects (spheres, cubes, cylinders, etc.). It's up to the programmer to synthesize complex models from the geometric primitives provided by OpenGL - points, lines and polygons.

The graphic utilities library (glu) provides commands (built using low level OpenGL commands) that perform tasks such as building parametric models, setting up viewing transformations and processing texture mipmap data.

In summary, OpenGL is an open, mature rendering library that is well supported and its advanced

[1] http://www.opengl.org

set of features offer programmers the opportunity of implementing advanced rendering techniques using optimised, platform independent code.

3.2 Three-dimensional rendering theory

One of the greatest challenges a graphic programmer faces when starting off is how turn a list of three-dimensional coordinates into an accurate visual representation using pixels on a screen. To answer this question, we have to harness both the tools provided by Euclidean geometry and those provided by OpenGL. In the following pages, I will highlight the most vital theoretical concepts but if you'd like to engage in a more detailed discussion, I refer you to [3] and [4].

3.2.1 Primitives and homogenous coordinates

OpenGL has only three geometric primitives: points, lines and polygons. These conform to their mathematic definitions except with regard to precision: computer systems use fixed precision floating point numbers which can give rise to rounding errors. Also, a pixel, the smallest graphic display element, has a finite size which does not conform to its mathematical definition of an infinitely small point. This leads to the situation where many points or lines with slightly different coordinates can be drawn to the same pixel(s).

All primitives are described as a set of floating point numbers called *vertices* (sing. *vertex*): one vertex for a point, two for a line[1] and *n* vertices for an *n*-sided polygon. Vertices are represented internally in terms of homogenous coordinates, which uses four floating point coordinates - *x, y, z* and *w* [2].

In OpenGL, polygons are areas enclosed by single closed loops of line segments, where each line segment is defined by their start and end vertices. A polygon can either be drawn filled in, as only line segments defining the sides of the polygon or as the vertices defining the corners of the polygon. In order to render polygons accurately, OpenGL applies the following restrictions to polygons:

i. The edges of polygons cannot intersect. This is then called a *simple polygon*

[1] In OpenGL, *line* refers to a line segment, as specified by the two vertices that constitute the start- and endpoints.

[2] If w is not zero, these coordinates correspond to the point (x/w, y/w, z/w). For more information on homogenous coordinates, see [1].

ii. Polygons must be *convex,* meaning that they cannot have indentations.

iii. Polygons may not contain holes[1]

Figure 1 illustrates valid and valid OpenGL polygons.

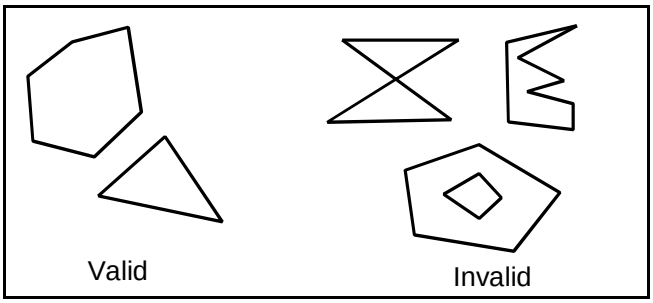


Figure 2: Valid and invalid polygons

These restrictions describe a class of polygons that can be drawn accurately and quickly and one often finds graphic adapters with hardware that accelerates the drawing of these polygons.

Since many surfaces fall into at least one of the above categories, we have to fragment these complex surfaces into a set of simple polygons. This process is known as *tessellation*. Another problem arises when using polygons with vertices that do not all lie in the same plane. By rotating these polygons,

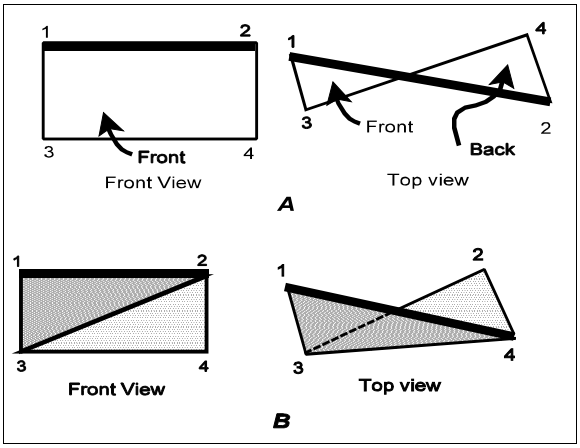


Figure 3: A) Nonplanar polygon transformed to nonsimple polygon by viewing from different angle; B) The same viewing transformation applied to the polygon tessellated using triangles

its possible that some of the line segments bounding the polygon cross over at some point, breaking

[1] This would require more than one closed line loop.

the simple convex polygon rule. We can avoid this problem by using triangles to tesselate surfaces, since triangles are the only polygons guaranteed to have all three vertices lying in the same plane. Figure 2 illustrates this condition, and its solution.

3.2.2 Polygonal vs parametric representations & resolution of polygonal models

Any curved or parametric surface can be approximated to some degree by polygonal facets. For example a cylinder can be approximated by the *n* polygons that represent the curved surface and the *2p* polygons that represent the end faces. The difference between the real and approximating surfaces (referred to as *piecewise linearities*) decreases as *n* and *p* increase, in other words as the polygonal resolution of the approximating model improves. But what advantage is there to be gained from approximation and just as importantly, what sacrifices do we have to make to reap these benefits? The most important advantage, especially with current and forthcoming graphic adapters, is hardware support for accelerated polygon rendering[1]. Also, modelling objects using polygons is straightforward and piecewise linearities can be minimised by using a shading technique (more on this later); Furthermore, geometric (as well as colour and texture) data is stored only for vertices and all other information is deduced from this data. The biggest disadvantage of polygonal models is their complexity - high resolution models often contain in excess of 100 000 polygons - and if a model was subject to continuous deformations the processing time required to reconstruct the model could easily push the performance levels below those required by interactive applications. Polygonal models also make texture mapping difficult to implement and increases the complexity of shadow algorithms.

3.2.3 Backface culling

Any enclosed polygonal surface can be seen to have two surfaces - an inner and outer surface. If the surface is opaque, only one surface is visible at any one time, depending on where the viewpoint is inside or outside the enclosing surface. *Backface culling* refers to the process of discarding polygons that are obscured by other polygons closer to the viewpoint.

3.2.4 Normal vectors

[1] Especially when using triangles; Gouraud shading - a fast shading algorithm based on polygonal representations - is also accelerated in hardware on modern graphics stations.

A normal vector is some vector that lies perpendicular to a surface. If that surface lies within a plane, then all its points on that surface have the same perpendicular direction and thus only one normal vector is necessary to describe all points within that surface. This is not the case for curved surfaces, where each point may require a different normal vector. In OpenGL, normal vectors can only be assigned at vertices.

Normal vectors are required by lighting calculations to determine how much light from a particular light source is reflected off the surface in the direction of the viewer. Also, bear in mind that any given point on a surface has two normal vectors and that they point in opposite directions. In fact, it is the direction of the normal vector that determines whether a polygon is *front-* or *back facing*.

Since normal vectors are used to indicate direction only, length is irrelevant. However, lighting calculations require vectors of unit length and so we first have to *normalise* our direction vectors by dividing each of the *x, y, z* components by the length of the vector, i.e.

$$(x_n, y_n, z_n) = \left(\frac{x}{l}, \frac{y}{l}, \frac{z}{l} \right), \text{ where } l = \sqrt{x^2 + y^2 + z^2}$$

Transformations such as scaling and shearing affect the length of vectors, so it is important to renormalise these vectors after such transformations and OpenGL provides such facilities. Calculating normal vectors is straight forward and is explained in detail in [2].

3.2.5 Constructing and viewing three-dimensional scenes

One of the biggest obstacles beginner in three-dimensional graphics has to overcome is learning how to specify both the positions and orientations of objects in a scene as well as the position of the viewer. Different *coordinate spaces* are used during the process of constructing a scene and we use three-dimensional *transformations* to move constructed objects from one coordinate space to another. A very useful analogy to keep in mind is that of a photographer taking a picture of still life scene he has constructed. So let's start at the beginning, in a world we call *object space*. Here, an object is constructed from primitives in terms of *object coordinates*[1] and bears no relation to other

[1] Object coordinates correspond to a default position, scale and orientation for all objects under construction. All objects share these defaults until moved to eye coordinates.

objects in terms of size, orientation or position. Once an object has been constructed, we apply rotations, scaling operations and translations until the object is in the right place, has the right size and is oriented correctly in relation to other objects in the scene. A *modelview matrix* is used to transform an object from object coordinates to *eye coordinates*, from object space to world space. This is analogous to the artist constructing his still life scene. Once the scene has been constructed, the artist has to set up his camera, i.e. choose his viewpoint. Essentially, the artist is creating a viewing volume dictated by what he sees through the lens of his camera. A *projection matrix* is used to specify such a viewing volume and transforms the scene from eye- to *clip coordinates*[1]. During this stage, *perspective division*[2] is also performed, thereby producing *normalised device coordinates*. Finally, the transformed coordinates are converted to *window coordinates* by applying the *viewport transformation* and is analogous to the photographer setting the magnification and dimensions of the photograph before printing it.

Although this might all seem intuitive, confusion arises when constructing the above mentioned matrices because of the order of operations. To specify modelling, viewing and projection matrices, we construct a 4x4 matrix **M** , which is multiplied by the coordinates of each vertex **v** in the scene to accomplish the transformation

$$\mathbf{v}' = \mathbf{M}\mathbf{v}$$

The confusion arises because matrix multiplications do not commute, i.e.

$$\mathbf{SRT} \neq \mathbf{RTS} \neq \mathbf{TSR}$$

where **T**, **S** and **R** refer to matrices that specify a translation, scaling and rotation respectively. So to perform a scaling, rotation and translation (in that order) to vertex **v**, we need the equation

$$\mathbf{v}' = \mathbf{TRSv}$$

Any other combinations of matrices would have a different result.

3.3 Vertex arrays

With polygonal models, one often finds that one vertex is shared by one or more polygons. A typical

[1] Objects (or parts thereof) lying outside the viewing volume are not visible and so are *clipped* from the scene by the boundaries of the viewing volume. These boundaries are referred to as *clipping planes*.

[2] Dividing coordinate values by *w*. See "Primitives and homogenous coordinates"

polygonal renderer will draw an object by sequentially rendering each of its polygons. This means that if vertex *a* is shared by *n* polygons, then *a* will be processed and drawn *n* times. To alleviate this redundancy, OpenGL provides data structures that store various aspects pertaining to an object's vertices (vertex and normal vector coordinates, colour data and texture coordinates). Then, during the rendering process, we call one command (instead of calling *n* commands for the *n* aspects we wish to specify) to extract and process all data specific to one vertex, polygon or object. Furthermore, OpenGL tries to cache previously processed data, thereby further cutting down on processing time. The programmer specifies whether to render one vertex at a time, the whole polygon or an entire object.

3.4 Display lists

First something on OpenGL's client-server model:
OpenGL is designed to work efficiently both in terms of local and distributed rendering. In a distributed environment, the machine running the rendering software (which issues the drawing commands)is termed the *client* whereas the machine receiving these commands (across a network) and rendering the picture is called the *server*. A standardised communication protocol exists between client and server, allowing client and server to exist on different platforms. With local rendering, one machine acts as both the client and the server.

In an attempt to minimise the traffic between server and client, OpenGL implements *display lists*. A display list is a cache of commands that, once compiled, is stored at the server. This means that if a program on the client wants to execute the commands stored in a display list, it need only send a single command across the network instructing the server to execute that specific list of commands, thereby minimizing network traffic.

Another advantage display lists offer is that of batch commands. Imagine we want to render a pyramid of 14 steel balls. Now, rendering a high resolution image of a metallic sphere is very expensive and doing so 14 times per frame would certainly reduce performance to below real-time figures. What display lists allow us to do is store only the results of the set of commands rendering a single sphere in a list and then duplicate these results at the position of each steel ball. The same applies to drawing the wheels of a car or drawing individual leaves of a tree.

3.5 Rendering contexts, capabilities and the framebuffer

A *device context* is a part of the operating system that interfaces with the device driver of a graphics device. The device context also stores information describing the graphic device's *capabilities* i.e. the rendering features supported by that device.

An OpenGL *rendering context* is a port through which all OpenGL commands pass. Every program that makes OpenGL calls needs to have a current rendering context, since contexts link OpenGL commands to the graphics device via a device context (see figure 3). The OpenGL rendering context also acts as a repository of state variables - for example, the current drawing colour or shading model.

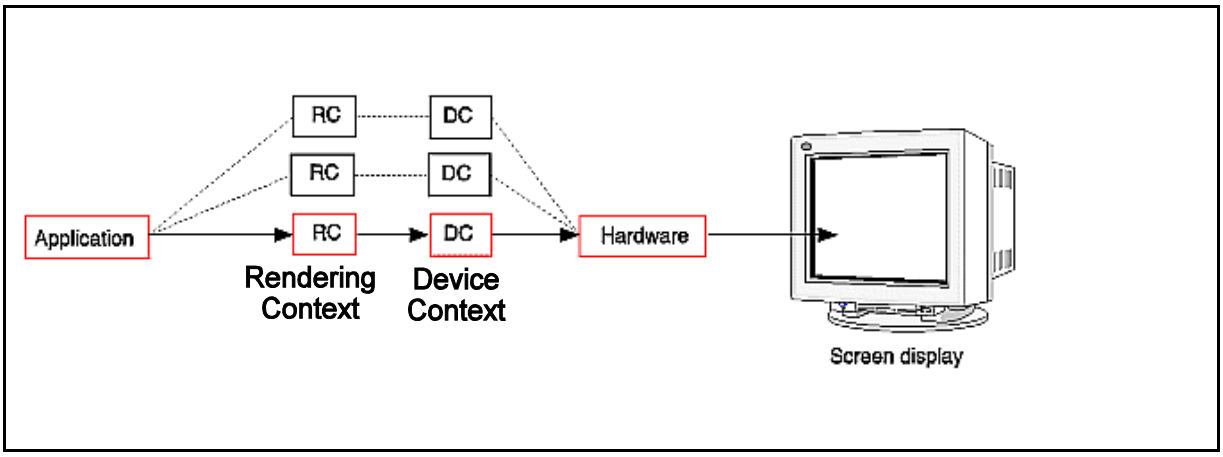


Figure 4 An OpenGL rendering context controls the graphics device by means of a device context. Each device context has a predefined set of features, described by its *pixel format*.

A *buffer* is a block of memory that stores uniform pixel data for all pixels of a screen. An OpenGL implementation typically supports the following types of buffers:

- Colour buffers (front-left, front-right, back-left, back-right and auxiliary buffers)
- Depth buffer
- Stencil buffer
- Accumulation buffer

The colour buffers are the most important, since this is where the final RGBA pixel representation of a picture is stored. Notice that there are four colour buffers. The reason why we have a front and back set is to allow *double buffering*. This capability allows flicker-free animations by first drawing the contents of the front buffers to the screen. While this is taking place, the next frames are rendered

to the back buffers. Then, when rendering to the back buffers is complete, the buffers are swapped, displaying the back buffers and allowing rendering to the front buffers to start. And so the process continues. Although double buffering produces smooth animations it limits the maximum frame rate to the monitor's refresh rate.

The reason for different left- and right colour buffers is to enable stereoscopic rendering: the image for the left eye will be drawn to the left buffer and the image for the right eye to the right buffer.

The depth buffer[1] keeps track of a pixel's depth in terms of eye coordinates. If an incoming fragment has a z coordinate closer to the viewer than the current pixel, the incoming fragment overwrites the current pixel and places its own z coordinate at that pixel location in the depth buffer. This process, known as *hidden surface removal*, ensures that only objects which are visible to the viewer are drawn. Disabling this depth test causes all surfaces to be drawn to the screen.

The stencil and accumulation buffers provide the graphics programmer with further drawing capabilities but since they were not used in this project, I will leave the reader to explore their uses in his own time.

The collection of buffers listed above is called the *framebuffer*[2]. When an application sets up its rendering context, it interrogates the windowing system to determine which graphic characteristics are supported and sets up the framebuffer accordingly. Thus if, for example, your implementation of OpenGL does not support stereoscopic rendering (do I see Microsoft fidgeting nervously?), you will find that the right-front and right-back colour buffers are not available.

3.6 The OpenGL pipeline

The OpenGL processing mechanisms that turn primitives and state information into a final picture is collectively called the *pipeline*. Those who've looked under the hood will agree that the pipeline is very complex piece of machinery, so what I propose to do in the following paragraph is give the

[1] Sometimes referred to as the *z buffer*, since it stores z coordinates

[2] Think of a frame containing a stack of buffers

reader a general overview of the pipeline that allows her to see how a primitive is turned into a pixel.

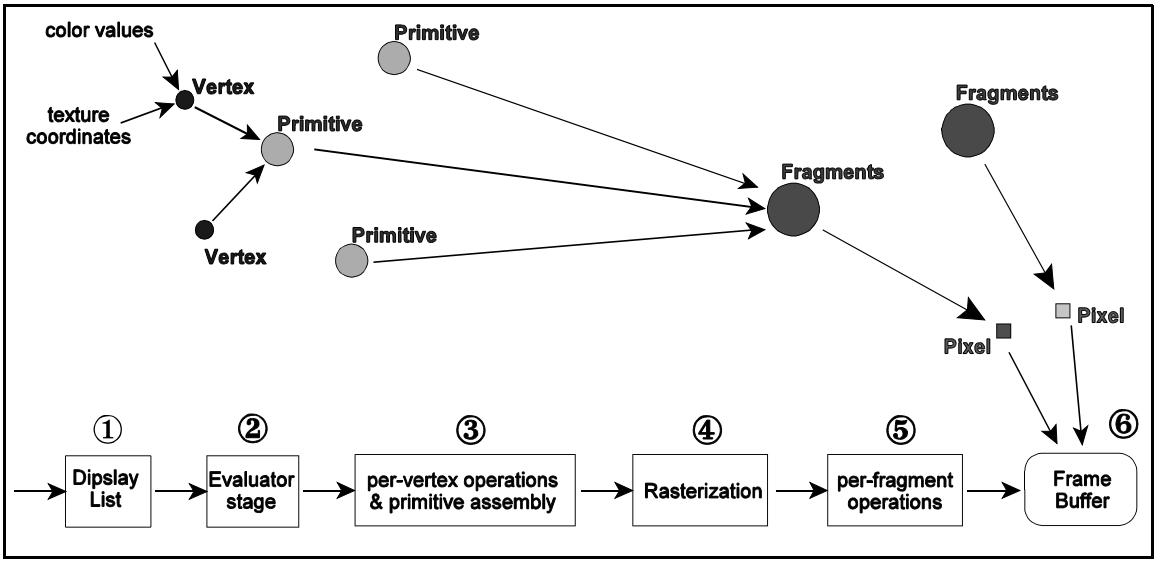


Figure 5: The OpenGL pipeline, illustrating how vertex data is processed to yield pixels

Figure 4 represents a simplified version of the pipeline.

All geometric primitives are eventually described in terms of vertices[1] and stages ① and ② deal with

loading vertex data from structures such as display lists or *evaluators*[2] into the first stage of the pipeline. Per-vertex calculations are performed on each vertex at stage ③, followed by rasterization to *fragments*[3] at ④. These fragments are subjected to a series of per-fragment operations at stage ⑤, after which the final pixel values are drawn into the framebuffer (step ⑥).

3.7 Colour

[1] In parametric representations, the data is converted to vertices and treated as vertices from then on.

[2] Evaluators are OpenGL calculations that calculate the object coordinate vertices given the object's parametric representation (typically a Bezier curve)

[3] Fragments are generated by the rasterization of primitives. Each fragment corresponds to a single pixel and includes colour and depth data as well as texture-coordinate data (if necessary)

For photorealistic images, colour and its interaction with the general light model of the scene is very important. This interaction is also very expensive, to the extent that none of the photorealistic renderers that I have come across can render at frame rates anywhere near those required by real time systems. This is the primarily why virtual reality systems do not strive for photorealistic scenes. Yet colour and lighting play vital roles in our perception of the environment and so it's important to render these two factors as accurately as possible.

At this stage I refer the reader to a good discussion regarding colour and lighting models in [5], since this is not really the place for an in-depth discussion of these topics. But let me highlight important aspects that relate to this project.

OpenGL provides us with two colour modes: RGBA and colour index. RGBA mode requires the programmer to specify separate values for each of its red, green, blue and alpha[1] values, whereas colour index mode only requires an index to a lookup table (palette) of fixed colours. The nature of the application will dictate which mode is eventually used. Working with a fixed palette may be easier but you pay the price of lost flexibility. In addition, many OpenGL functions do not support colour index mode.

3.8 Shading

Shading an object plays a vital role in providing the viewer with perceptual information regarding the three-dimensional structure of an object and the lighting environment that that object finds itself in. OpenGL offers the programmer two shading modes: flat shading and smooth shading. With flat shading we draw an entire primitive using one colour. With smooth shading, we treat the colour at each vertex of the primitive individually and interpolate between these colours for points between the vertices (for lines) or within the polygon. How this interpolation is performed is a whole research area in itself, but for real time systems, two main algorithms stand out: Gouraud and Phong shading. These methods produce different results but both are aimed at efficiently shading polygonal meshes[2]. This means that the models do not only try to convey the three-dimensionality of the object but also try to shade the object in such a way that the piecewise linearities of the approximating polygonal

[1] Alpha values will be covered when discussing blending

[2] Neither Gouraud nor Phong shading can be applied to non-polygonal models

model are rendered invisible. The intensity of (reflected) light is calculated as a function of the vertex normal with respect to both the position of the light source and the eye.

With Gouraud shading red, green and blue intensities are calculated for each vertex of the model and then linearly interpolated for values between the vertices. The vertex normal used to calculate these intensities is calculated such that it approximates the normal to the original, unpolygonalized surface at that point. We do this by taking the normalized average of the normal vectors of all the polygons that share the vertex in question. The success of the Gouraud algorithm depends largely upon the surface being relatively smooth - sharp surface features are often lost because they are averaged out by neighbouring polygons. By introducing weighted averages, as discussed in [6], we can obtain better results for Gouraud shaded models.

Because Gouraud shading only evaluates the light intensities at the vertices, it does not handle highlights very well. Consider the case where a tight beam of light intersects the centre of a polygon. Since the beam does not spread enough to cover one of the vertices, its contribution to the intensity calculations are zero and when the interpolation takes place, the specular highlight (to be described shortly) that should be seen in the centre of the polygon is not rendered.

Another defect of the Gouraud algorithm is Mach banding. This is a psycho-physical phenomena that causes light bands to be perceived at polygon boundaries - even though there is no colour discontinuity across the boundary. Gouraud himself explains the phenomena as follows:

> "The linear interpolation which has been used here produces a shading which is continuous in value but not in derivative[1] across polygon boundaries. The resulting Mach band effect can be observed mostly in the vicinity of silhouette curves and where the surface bends sharply."

The only difference between Gouraud and Phong shading is that, instead of interpolating intensities, Phong shading interpolates the *vertex normals* and then evaluates the RGB intensity *at each pixel*. This introduces substantial processing overheads but it does produce more accurate results (especially in terms of specular highlights) and also minimises Mach banding.

---

[1] The human visual system enhances the second derivatives of intensity changes to sharpen the perception of object's edges

In terms of performance, the extra calculations required to Phong shade a model leads to slower performance than that obtained by Gouraud shading. Furthermore, many modern graphics accelerators provide hardware accelerated Gouraud shading and I suspect that this is why it is the default smooth shading algorithm supported by OpenGL. Alan and Mark Watt provide a more in-depth discussion of shading algorithms in [7].

3.9 Lighting

When working with a lighting model, there are two aspects that we have to consider: the nature of the light source and that of the material properties of surfaces that will reflect the incident light. Let's look at the source first. Light incident on a surface can be either ambient, diffuse or specular in nature.

3.9.1 Light source properties

Ambient illumination is light that has been so scattered by reflections off surfaces that its original direction cannot be determined. It is effectively directionless and OpenGL approximates it by assuming that it comes from all directions.

Diffuse illumination is light that comes from a single direction. Thus, if it intersects a surface perpendicularly we get the maximum amount of reflection; if the surface lies parallel to the direction of the light, no light will be reflected. The name diffuse refers to the way the incident light is scattered in all directions when reflected, so the reflection is equally bright from all angles.

Specular light is directional both in terms of the incident and reflected light. This means that, unlike, diffuse light, the intensity of a specular reflection varies considerably as the viewpoint changes.

The OpenGL lighting model consists of a single ambient light source and a variable number of independent lights, each with its own ambient, diffuse and specular components. The maximum number of lights available to the programmer depends of the implementation used. The ARB specifies that for implementations complying with the standard, at least eight light sources should be supported.

The OpenGL lighting model approximates light by assuming that it can be composed of a red, green and blue component. Each component is described by its intensity, a real number between 0.0and 1.0. A value of 0.0 indicates that there is none of that component present in the emitted light, whereas a value of 1.0 indicates that that component is displayed at the device's full intensity. Here is a table illustrating some key colours and their component intensities:

Colour	Red	Green	Blue
Red	1.0	0.0	0.0
Green	0.0	1.0	0.0
Blue	0.0	0.0	1.0
White	1.0	1.0	1.0
Black	0.0	0.0	0.0
Yellow	1.0	1.0	0.0

The net effect of a light is composed of a combination of the ambient, diffuse and specular properties of that light. OpenGL allows us to set the red, green and blue values of each component separately. For example, if we have a white light illuminating a blue sphere and the light scattered off the sphere is blue, then we are seeing the blue diffuse component of the light. If our light has a strong specular component, we will see a white specular highlight on the blue sphere.

3.9.2 Position and attenuation of light sources

As hinted before, there are both directional and non-directional light sources. A *directional* light source has light rays that are parallel - essentially the effect we get with sunlight - whereas non-directional, or *positional* light sources, have light rays that diverge in all directions from a local point source (e.g. candle light). By default, positional light sources radiate in all directions, but OpenGL allows us to confine a directional light source to a cone by specifying it as a spotlight.

For real world lights, the intensity of the light decreases as we increase our distance from the light. This property, also known as *attenuation,* is calculated using the "Inverse Square Law":

$$\mathbf{I}_{eye} = \left(\frac{1}{\mathbf{k}_q d^2} \right) \mathbf{I}_{source}$$

where

$\mathbf{I}_{eye}$ is the intensity of the light reaching the eye,

$\mathbf{I}_{source}$ is the intensity of the light source,

$\mathbf{k}_q$ is the quadratic attenuation constant and

d is the distance of the eye from the light source.

OpenGL uses a modified version of the above equation to allow for constant, linear and quadratic attenuation:

$$I_{eye} = \left(\frac{1}{k_c + k_l d + k_q d^2} \right) I_{source}$$

For directional lights, OpenGL does not apply attenuation and only the direction of the light source is specified. For positional light sources, life is a bit more complicated and we have to specify both the position and attenuation. A light's position is specified in homogenous coordinates. This means that all transformations applied to coordinates of objects in the scene can also be applied to coordinates describing a light 's position. This allows us to translate and rotate lights[1] in the same way we manipulate other objects in the scene. If a positional light is designated as a spotlight, we additionally have to specify the direction and cut-off angle of the spotlight. Figure 5 illustrates the three types of lights described so far.

[1] It is pointless trying to scale a point source (no pun intended).

3.9.3 The viewpoint

The location of the viewpoint affects the calculations for highlights produced by specular reflection. More specifically, the intensity of the highlight at a particular vertex depends on the normal at that vertex, the incident angle θ of the light reflected off the object and the angle ϕ between the vertex normal and the viewer. With an infinite viewpoint, OpenGL assumes ϕ is constant for all vertices. For a local viewpoint, OpenGL calculates ϕ for each vertex. This produces more realistic results but is more computationally intensive.

3.9.4 Two-sided lighting

OpenGL performs lighting calculations on both front- and back-facing polygons. The programmer can choose whether to process back-facing polygons in the same way as front facing polygons are processed. OpenGL reverses the vertex normals for that polygon and processes it as usual. In addition, we can specify different material properties for back-facing polygons to those used for front-facing polygons. This is often useful when sections of an object are cut away to reveal the interiors, revealing the back-facing polygons that are never seen from the outside.

3.9.5 Material properties

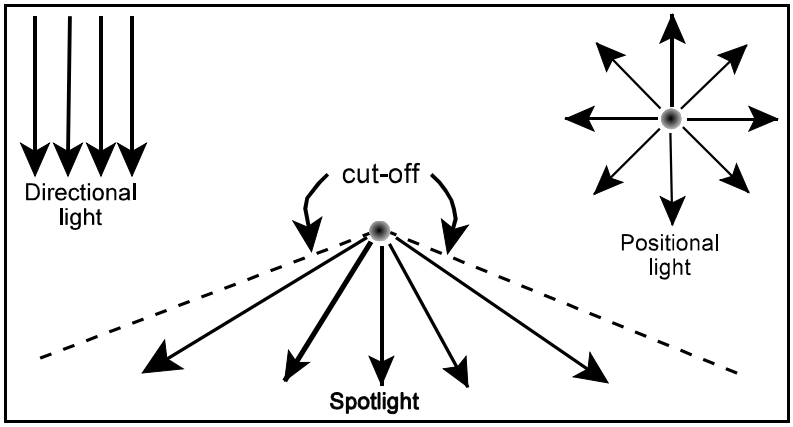


Figure 6: OpenGL supports directional lights, positional lights and spotlights.

In OpenGL there is a very strong interaction between light and material properties in order to

determine the final RGB colour written to a pixel. Material properties specify the reflective and emissive properties of a surface. In our example above of a sphere, only the blue part of the diffuse component is reflected because the material absorbed the red and green components. A white highlight was reflected because the material reflected all three components of the incident specular light. When specifying material properties, we specify what proportion of the red, green and blue incident light is reflected: a value of 1.0 signifies complete reflection whereas a value of 0.0 signifies complete absorption. Again, OpenGL allows us to specify the RGBA values for each property independently. The emissive property works differently: it specifies how much light is *emitted* by a surface. This emitted light is unaffected by other light sources and introduces no further light into the scene, i.e. an emissive surface is not a light source in the OpenGL sense.

This interaction between light and material to produce a final colour can be represented mathematically as follows. Say we want to compute the diffuse light reflected off a surface and reaching the eye. Let the diffuse components of the light source be represented by L_r, L_g, and L_b for the red green and blue components respectively. Likewise, the diffuse material properties are represented by S_r, S_g and S_b. The resultant diffuse light reaching the eye is then given by the equation

$$E_{diffuse} = L_r S_r + L_g S_g + L_b S_b$$

$E_{ambient}$ and $E_{specular}$ are calculated in the same way.

3.9.6 The role of Normal Vectors

A polygon's normal determines the direction the polygon faces. This is why OpenGL uses the vertex normals to calculate how much light is reflected off the polygon in the direction of the viewpoint. Remember that lighting calculations are only applied at vertices and using vertex normals. In Gouraud shading, the intensities are interpolated, whereas in Phong shading it is the normals which are interpolated. Irrespective of which shading model is used, we are bound to get more accurate lighting results by increasing the polygonal resolution of the model. This minimises the interpolation between any two points on a surface, resulting in more accurate highlights and shading gradients.

3.10 Blending

In addition to specifying the red, green and blue components of a colour, OpenGL allows us to specify an *alpha* value which is used in blending calculations. The alpha value is used to combine

the fragment being processed with that of the pixel already stored in the framebuffer. This comparison takes place after rasterization but before the final pixels are drawn to the framebuffer.

The easiest way to approach blending is to think of the red, green and blue components of a fragment as representing its colour and the alpha component representing opacity. Transparent or translucent surfaces have lower opacity than opaque ones and hence lower alpha values. As with RGB values, alpha values lie in the range [0.0, 1.0]: 0.0 indicates full transparency whereas 1.0 indicates full opacity. There are many ways to calculate the net effect of various colours blended but I will only discuss the algorithm used later here.

Suppose we have n overlapping pixels (p_1, p_2, ..., p_n) in the scene and each of these has an associated alpha value, a_n. (p_1 has alpha value a_1, p_2 has alpha value a_2, etc.). Say that pixel p_{fb} has already been drawn to the framebuffer and we now have to draw pixel p_n. The net colour of the pixel finally drawn to the framebuffer, p_{new}, is given by

$$p_{new} = p_{fb}(1.0 - a_n) + p_n a_n$$

Note that blending is applied to one incoming pixel at a time. Thus, if there are n overlapping pixels in the scene, the above equation will be applied n times.

I think an example will help quantify the theory. Imagine we have three coloured planes, p_1, p_2 and p_3. The following table summarises their RGBA values:

Components:	**Red**	**Green**	**Blue**	**Alpha**
$\boldsymbol{p_1}$	0.0	0.0	1.0	0.3
$\boldsymbol{p_2}$	1.0	0.0	1.0	0.6
$\boldsymbol{p_3}$	0.5	0.8	1.0	0.5

The first time we have to apply the blending calculation is when p_2 needs to be blended with p_1. Then

$$\begin{aligned} p_{new} &= (0.0,0.0,1.0)(1.0-0.6) + (1.0,0.0,1.0)(0.6) \\ &= (0.0,0.0,1.0)(0.4) + (1.0,0.0,1.0)(0.6) \\ &= (0.0,0.0,0.4) + (0.6,0.0,0.6) \\ &= (0.6,0.0,1.0) \end{aligned}$$

Next, p_3 is blended with p_{new} to give the final pixel:

$$
\begin{aligned}
p_{final} &= p_{new}(1.0 - a_3) + p_3 a_3 \\
&= (0.6,0.0,1.0)(1.0 - 0.5) + (0.5,0.8,1.0)(0.5) \\
&= (0.3,0.0,0.5) + (0.25,0.4,0.5) \\
&= (0.55,0.4,1.0)
\end{aligned}
$$

Thus, the final components of the pixel in the framebuffer are red = 0.55, green = 0.4 and blue = 1.0.

3.11 Texture mapping

Anyone who has experience building scale models will know what a blessing *decals* are. These little stickers (which are glued to the surface of the model) saves the modeller hours of painstaking brushwork. And the good news is we can use the same technique in computer graphics to approximate surface features of an object - it's called *texture mapping*. More specifically, texture mapping is the process of taking a two-dimensional picture (the texture) and mapping it to an OpenGL primitive (usually lines and polygons). Using texture mapping, we can approximate the visual characteristics of a surface instead on building an accurate, but complex, model that produces the correct result when adequately lit. Furthermore, the OpenGL texture mapping model ensures that the same transformations that are applied to geometric primitives are also applied to textures. This ensures that, as the model undergoes transformations, the textures mapped to its surface undergo the same process, ensuring accurate visual results. Nevertheless, texture mapping is a complex subject, since we are taking a two-dimensional picture, fragmenting[1] it and pasting each fragment onto a three-dimensional polygon. In addition, the programmer has to specify how the texture interacts with the other surface properties and how textures are warped to maintain perspective, amongst other things.

Fundamentally, textures are two-dimensional arrays of colour data and each element in such an array is called a *texel*. One of the biggest challenges is taking a group of texels and mapping it to a polygon

[1] This does not refer to fragments generated by the rasterization stage of the OpenGL pipeline

in such a way that the aspect ratio is preserved and the image looks correct, once mapped. This is not a trivial problem. Furthermore, transformations may result in more than one texel being mapped to one fragment or that one texel covers more than one fragment. We then have to employ *filtering* techniques to map texels to fragments, either by averaging the multiple texels mapped to one fragment (this is called *minification*) or by enlarging a texel to map to an entire fragment (*magnification*). These mapping and filtering operations make texture mapping a computationally intensive process, which is why many of today's graphics cards have dedicated hardware support for texture mapping.

But how do we actually map the texture onto the polygon? To accomplish this, we provide the *x* and *y* coordinates of the texture map where the vertex intersects the texture map and call these coordinates the *texture coordinates*. These are traditionally referred to as *s* (instead of *x*) and *t* (instead of *y*). These coordinates are provided for each vertex of the polygon and each coordinate lies in the range [0.0, 1.0], where (0.0, 0.0) is the bottom left corner of the texture map and (1.0, 1.0) is the top right corner.

3.11.1 Texture objects

In order to manage the large amounts of resources required by texture mapping, OpenGL 1.1 introduced *texture objects*. Each texture object is responsible for managing a single texture, the parameters used to map that texture and associated mipmaps (discussed shortly) derived from that texture. To further enhance performance and memory management, certain implementations of OpenGL support a working set of texture objects, referred to as *resident* textures, that exhibit better performance than texture objects outside the working set.

3.11.2 Mipmapping

Imagine we have an animation of Cool Hand Luke, the cowboy, riding his horse off into the sunset. In order to get the checkered pattern on Luke's shirt, we used a texture map. Now, in order to maintain the perception of perspective, the texture map needs to shrink as Luke and his shirt get smaller and smaller. OpenGL offers the programmer two ways of shrinking a texture: minification

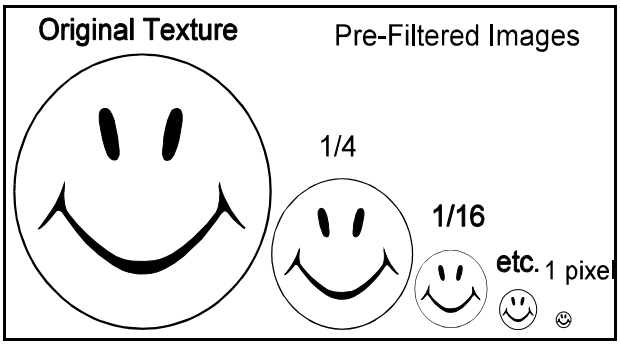


Figure 7: When supplying mipmaps, they have to be in sizes of powers of 2 of the original image.

filtering and mipmapping[1]. Using filtering would mean taking averages iteratively and often leads to disturbing visual artifacts due to errors propagated and compounded by the averaging process. However, by using a set of prefiltered texture maps of decreasing resolutions we can improve the accuracy of the filtering process. This process is called *mipmapping*. OpenGL automatically determines which texture map to use based on the pixel size of the object being mapped. Thus, when our animation starts and Luke is close, OpenGL will use the highest resolution texture map. As Luke rides off into the sunset, OpenGL will switch successively to lower resolution texture maps and use these to produce the checker pattern on Luke's shirt.

When supplying texture maps, we provide all sizes of the original texture in powers of 2 between the largest (original) size and a 1x1 map. Figure 6 illustrates these multiple textures.

3.12 Display lists, texture objects and rendering contexts

And important (and undocumented) point that we discovered when rendering with multiple, simultaneous rendering contexts is that texture objects and display lists are context sensitive. This means that if these objects are created when one rendering context is current, they are not available when switching to another context. This is due to the client-server architecture of OpenGL, where the display list resides on the server. Since each server will have a different rendering context, the various rendering contexts could well be widely distributed geographically. In this case it is better to send each server its own display list that could be stored locally. If rendering contexts could share display lists in such a scenario, we would have display lists flying all over the network and that sort of thing makes network administrators *very* nervous. As a result, any display list used by a rendering

[1] *Mip* stands for the Latin "*multim im parvo*", meaning "many things in a small place".

context has to be compiled when that context is current. The same seems to hold true for texture objects. As you will see later, this affects our VisualRepresentation component when it compiles its display list.

3.13 Stereoscopic rendering

When rendering a scene in stereo, we produce two images, called *stereo pairs*. One image is drawn from the point of view of the left eye and the other from the point of view of the right eye. The difference in the perceived scenes - called *binocular disparity* - produces an important depth cue, called *stereopsis* (cf [8]). These two images can then be displayed by a suitable display device (head-mounted display, shutter glasses or using red and green filters).

Figure 7 illustrates the technique used to generate the correct picture for each eye. To get the right-eye image, we translate the original scene by half the interocular distance, *d*, to the left as shown at ① and then render the scene. We then translate the scene *d* units to the right to get the left-eye point of view and render to get the left eye image (as shown at ②). ③ shows how *d* is half the distance between the viewer's eyes.

There are various techniques used to display stereo images. The first and probably the simplest is

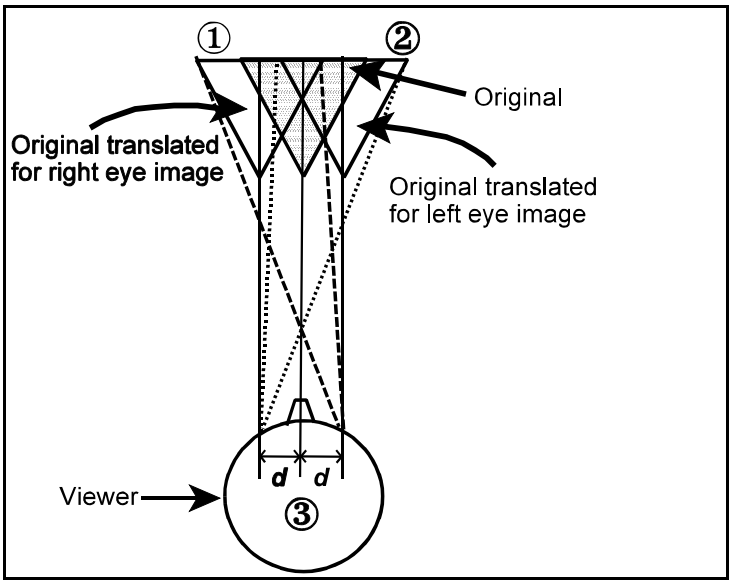


Figure 8: In order to generate the image for the left eye, we translate the original scene ***d*** units to the right, and *vice versa* for the right eye.

the red/green filtering technique. Here, the left-eye image is drawn in green and the right-eye image in red. The user then wears a set of glasses with a red filter in front of the right eye and a green filter in front of the left eye. Each filter will then transmit only that part of the picture which is destined for that eye and absorb the rest. This approach does not require specialised hardware, since both images are displayed on the same output device. Its disadvantage is that each of the images is restricted to the colour destined for that eye This means that we are effectively rendering in monochrome.

Another technique that uses the same output device is called *interlacing*. Here we use the scan lines of the device to alternately display the lines of the left and right images. The disadvantage of using interlacing is that, to fit both images onto the screen, each image needs to be half the height of the output device. This requires adjustments in the aspect ratio. Another disadvantage is that setting up interlacing for a monitor requires platform dependant calls to the windowing system. This could make interlacing an unattractive option for systems that render on remote machines.

A technique which does rely on specialised hardware is *shuttering*. Here, the viewer looks at the monitor through a pair of LCD goggles. The goggles act as electronic shutters that alternate which eye is exposed to the monitor. As the left-eye image is displayed, the right lens will enable its

shutter, blocking out the image. Likewise, the left-eye shutter will engage when the right-eye image is displayed. An interface card synchronises the shutters with the frames displayed by the application.

Head-mounted displays (HMDs) also rely on dedicated hardware but instead of controlling which eye receives the image, it uses a dedicated display for each eye. The two displays, together with corrective optics and other electronic devices are assembled into a compact unit that is worn on the user's head. HMDs provide the greatest immersive effect but are also the most expensive of the stereographic rendering techniques. Another shortcoming shared by shuttering and HMDs is that it requires each frame to be drawn twice - once for each eye - effectively cutting the frame rate of the application in half.

3.14 Optimisation

As the reader might have guessed, maintaining real-time frame rates[1] when rendering complex scenes is no small feat. For this reason, designing and implementing optimal rendering routines has been one of the prime objectives of this project. In fact, one of the main reasons for choosing OpenGL as our rendering API has been the large range of optimisations it offers the programmer. These optimisations are also well documented and flexible enough to be applied to a wide range of applications. Two of these mechanisms, display lists and vertex arrays have already been discussed. The following paragraphs discuss the various stages of the OpenGL pipeline that are prone to bottlenecks and how we can optimise rendering performance by alleviating these bottlenecks. I then go on to discuss the concept of feedback optimisation and how this mechanism tries to maintain real-time rendering by dynamically adjusting the capabilities used during the rendering process.

There are three essential stages of the pipeline that are prone to bottlenecks:

1. The CPU subsystem
2. The graphics subsystem
3. The rasterization subsystem

At any one stage, one of these stages can become a bottleneck if overloaded by the application. The online Iris Insight library [10] found on the SGI machines offers formal techniques for identifying

[1] The MPEG consortium specifies 25 frames per second as the minimum frame rate required for smooth animations

and reducing bottlenecks by distributing the graphics load more evenly across the three stages.

3.14.1 Principles of feedback optimisation

As you will see later when I discuss performance results, the frame rate of an animation relies heavily upon the complexity of the scene being drawn. The number of polygons being rendered, the use of smooth shading, blending, texture mapping, texture filtering and colour rendering all combine to decrease the frame rate of an animation. If you are fortunate enough to own a high performance SGI graphics workstation with plenty of hardware acceleration, maintaining a high frame rate is not really a problem. Working on a normal PC, however, is a different story and maintaining even a 25 f/s frame rate becomes a substantial challenge. The aim of performance *feedback optimisation* is to maintain a frame rate specified by the application programmer by dynamically varying the capabilities used during the rendering routine. In other words, we try to maintain a smooth animation by varying the quality of the final image. Figure 8 illustrates the feedback cycle.

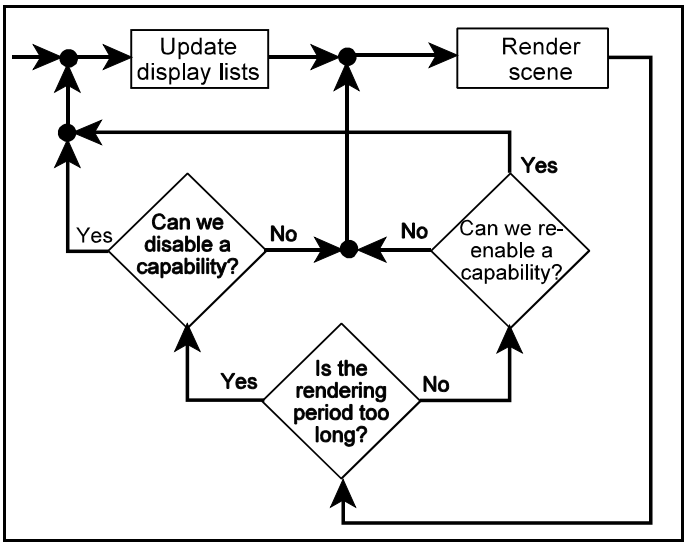


Figure 9: The feedback mechanism will switch capabilities on and off according to the frame rate.

The feedback mechanism constantly monitors a moving average of *n* frame rates. If that average falls below the frame rate (the *performance index*) specified by the programmer, the mechanism will disable the capability at the top of the capability stack. This process will be repeated after each cycle until the frame rate is restored to a value greater than or equal to the performance index. If the average after *n* frames is greater than the performance index, the feedback mechanism tries to re-

enable capabilities one at a time until the application is rendering with full capabilities once more. In static scenes, one tends to find that this mechanism either settles down to a stable value close to the performance index or oscillates around it. However, in dynamic scenes, the feedback is much more active in order to compensate for the greater variations in processing required to render the changing scenes.

Choosing *n*, the number of frames over which the average frame rate is evaluated (called the *sample period*), is an important decision: if *n* is too low, the mechanism will recompile the display lists too often, thereby incurring the overhead of compiling these lists and retarding performance; choosing *n* too large introduces significant hysteresis, or lag, into the feedback response and results in the mechanism reacting to a change in frame rate long after that change has passed.

4

Design and Implementation

Now that we have covered in detail the theoretical background required, it is time to start describing how the system was implemented. The LEGO paradigm of creating a complex system out of smaller components that work together naturally led to an object oriented approach. Thus, in the pages that follow, I will discuss the system in terms of its components, explaining issues relating to the purpose of each component, its design and the final implementation. Here's a brief overview of the components I will discuss:

- The **VROutputDevice,** which is responsible creating a platform independent rendering context as well as managing the three-dimensional rendering volume to which our application will draw;
- The **VRSink**, which is responsible for constructing the VR scene in a rendering context. It also instantiates the VRSinkMonitor and VRCapabilities objects , sets performance thresholds and enables feedback optimisation.
- The **VRSinkMonitor** component, which implements feedback optimisation.
- The **VRCapabilities** component, which keeps track of the status of the capabilities currently being used to render the scene.

- The **TextureManager** component, which creates and manages texture objects of all the texture maps used by the application.
- The **VisualRepresentation** component, which is responsible for inputting and parsing object data, synthesizing the visual representation and rendering this representation using enabled capabilities.

These six components, their auxiliary components and the interactions between components is illustrated in figure 9.

4.1 The Virtual Reality Output Device (VROutputDevice)

The primary objective of this component is to create and manage the rendering contexts used by the rest of the application. Ideally, the class should provide the rest of the system with a set of device independent methods that manage one or more rendering contexts, the associated viewports and the three-dimensional viewing volumes. The component also handles screen resize events and, if more than one rendering context exists, provides methods that perform context switches between the different contexts.

The class performs many of its duties by calling specific glut routines. However, this has its drawbacks: the glut library is aimed at small to medium applications and its author warns against using it for larger or more complex applications. The main reason for this is that glut implements a continuous loop, the **glutMainLoop**, that takes over the application completely once the rendering context has been set up. This loop contains event handling routines that call special *callback*[1] functions that the programmer has to define. This restricts the options of the programmer immensely, since the entire rendering routine has to be specified as one procedure called display(). This make it nearly impossible to build an application that uses multiple threads. To remedy this situation, we modified the glut source to remove the infinite loop. This gives the programmer the freedom to specify such a loop himself and tailor its behaviour to the specific application.

Despite these drawbacks, the glut library is flexible and stable and, as of version 3.7, can negotiate rendering contexts on both Unix and Win32 platforms. This means that we can use the high level (and platform independent) glut calls to create rendering windows and perform platform dependent

[1] These are functions called in response to run-time events.

tasks such as creating device contexts, setting pixel formats, handling buffers, etc. In fact, many of the window-related services offered by the VROutputDevice class are merely abstract interfaces to related glut calls.

The reason for using these abstract interfaces is polymorphism. This offers the programmer the opportunity to override the original methods with code that either uses more sophisticated features of future glut releases or doesn't use the glut library at all. This is important for porting the system to different platforms, since glut has not been ported to all operating systems and languages. Thus, if a programmer uses the windowing system's API to implement the services advertised by the VROutputDevice interface, these implementations are confined to the VROutputDevice and should be transparent to the rest of the system. This should make porting the system to other platforms substantially easier.

Lastly, let's take a look at the methods that implement the services offered by the VROutputDevice component:

VROutputDevice : the constructor

The component constructor takes as parameters the caption, position and size of the desired rendering window. First it initialises glut with a call to **glutInit().** Once the library is initialised, the constructor passes the parameters to the **glutInitDisplayMode(), glutInitWindowPosition**() and **glutInitWindowSize**() calls. Lastly, the callback functions are registered using the **glutDisplayFunc**(), **glutReshapeFunc**() and glutSetWindow() calls.

~VROutputDevice

The destructor release the device context and destroys the OpenGL rendering context by calling the **glutDestroyWindow**() procedure.

setWindowTitle

Sets the window caption by using the **glutSetWindowTitle**() procedure.

setWindow

Performs a context switch to the rendering context for this window. Calls **glutSetWindow**().

setDefaultWindow

This method makes current the first rendering context created using the glut constructor. Calls **glutSetWindow**().

setGlut

This method calls the modified glutMainLoop procedure, **glutMainTick**(). This executes all commands contained within a single iteration of the glutMainLoop, such as event responses and user input processing.

swapBuffers

This method causes the contents of the back buffer to be copied to the front buffer (and hence displayed). Calls **glutSwapBuffers**().

setProjection

Uses OpenGL calls to set up the viewpoint projection matrix. This can be set to either perspective or orthographic projection using the glFrustum() or glOrtho() commands, respectively.

4.2 The Virtual Reality Sink (VRSink[1])

The VRSink component has two main tasks

- instantiating and controlling a rendering window in the form of a VROutputDevice component
- construct the virtual scene using transformations

The VRSink class is inherited from the VRComponent class and is the component that draws the VR scene to the output device. The VRSink and VROutputDevice should be seen as an atomic unit that is a fundamental building block for any virtual reality application. The VRSink component calls the methods of the VROutputDevice it instantiates. Thus the VROutputDevice can be seen as providing low-level services that are utilized by the VRSink to draw the scene and manage the display window. The sink constructs this scene by using transformations (translations, rotations and

[1] Also referred to simply as “sink”

scaling operations) to move objects from object coordinates to eye coordinates.

One important point to realise is that the point of view will affect the translations used to locate an object in the scene. For example, the observer is looking at a cube rotating clockwise and moving left across the scene. However, from the point of view of the object, it is rotating counterclockwise and moving right across the scene. So it's all a matter of relativity. To handle this predicament, the sink interrogates the VREnvironment component to get the absolute position and orientation of the entire scene, gets the inverse of these values and applies them as transformations. This produces the correct scene relative to the current viewpoint.

The sink has been designed so that it is easy to create descendant components that cater for specialised output devices. This is the case with the VRStereoSink component. This component implements stereo rendering by adding another VROutputDevice component and overriding the ThreadRoutine method. The original ThreadRoutine takes care of rendering the scene from a specific viewpoint; VRStereoSink::ThreadRoutine renders the scene for each eye and introduces the correct transformations to produce the binocular disparity that is interpreted by the brain to provide the required perception of depth.
Currently, two separate windows are used to render the left and right images, despite many OpenGL implementations providing left and right buffers exactly for this purpose. The reason two separate output devices is used is that it would allow us to send the output of each window to a separate output device. These devices would then source the separate displays of the HMD. Whether this approach is feasible remains to be seen.

In terms of optimisations, the reader will find very few in the VRSink component. The reason for this is two-fold: firstly, none of the operations that typically cause performance bottlenecks are found in the sink; secondly, our only possible optimisation would be to compile the entire scene as a display list. However, since the scene is usually dynamic in animations display lists are not an option since the transformations required to place the objects in the scene change from scene to scene. If our scene is static, or if only the viewpoint changes between frames, then compiling the scene as a display list could prove to be a valuable optimisation.

Now lets take a look at the VRSink component's method:

VRSink: the constructor

This is where the VROutputDevice is instantiated and all the basic parameters determining the nature of the scene are set. The order of events are as follows:

1. Instantiate VROutputDevice
2. Set the current rendering context to that of the device context just created by VROutputDevice's constructor
3. Enable various OpenGL features:
    a. Depth buffering
    b. Auto-normalisation of surface normal vectors
    c. Blending
    d. Processing of material properties
    e. General ambient lighting
    f. Light0 - the single directional light source
4. Set various states that will affect different stages of the OpenGL pipeline:
    a. The blending function is set to use the function given in the Background section on alpha blending
    b. The Shading model is set to smooth shading
    c. The RGBA values for the ambient lighting are set
    d. Lighting calculations are set to use local viewpoints and to process only front facing polygons
    e. Light0 is set up:
        i. Position is set to [x, y, z, w] = [10,000.0, 10,000.0, 10,000.0, 1.0]
        ii. Ambient, diffuse and specular values are all set to white
        iii. The attenuation factors are set to use constant attenuation, $k_c = 1.0$
    f. Light0 is enabled
5. The current rendering context is set to the default window (the first VROutputDevice created)

setViewPoint

This is an accessor method that allows the application programmer to set the viewpoint by passing

a view vector as the parameter.

enableReport

This method instructs the VRSinkMonitor to write performance statistics to the console.

disableReport

This method disables the VRSinkMonitor's performance statistics from being written to the console.

setPerformance

This accessor allows the user to set the minimum frame rate required by the application. If a value of zero is passed, feedback optimisation is disabled; otherwise feedback optimisation is enabled and tries to maintain the specified frame rate. By default feedback is disabled - only a call to setPerformance will enable the mechanism.

getPerformance

This method returns the minimum frame specified by the user.

ThreadRoutine

This is the method that actually constructs and renders the scene. The sequence of events are as follows:

Firstly, we set the rendering context to the current window, clear the colour and depth buffers and set the projection transformation to use the perspective projection matrix.

The next group of operations sets up the transformations that will move objects from object to eye coordinates, as described above. Since matrix multiplications do not commute, we have to specify the transformations we wish to use in reverse so that the transformation that needs to be applied first is executed just before the object is drawn. Thus, the first transformation we encounter (but which will be applied last) is the viewpoint transformation. This calls the GetAbsolutePositionAndOrientation method to get the viewpoint vector and orientation, inverts

them and applies first the inverted translation and then the inverted rotation.

The next step is to construct the scene, i.e. scale, rotate and translate each object to its correct size, orientation and position in the world. For each object, a call is made to the GetWorldPositionAndOrientationAndScale method of the VREnvironment component to extract scale, orientation and positional data regarding the current object. These are then implemented as a translation followed by a rotation and then a scaling operation. Finally, the Render method of the current VisualRepresentation is called to render the object. When VisualRepresentation::Render is called, it is passed a parameter containing the active status of each rendering capability. This is obtained using a call to VRCapabilities::getCapabilities method. This process is illustrated in figure 10.

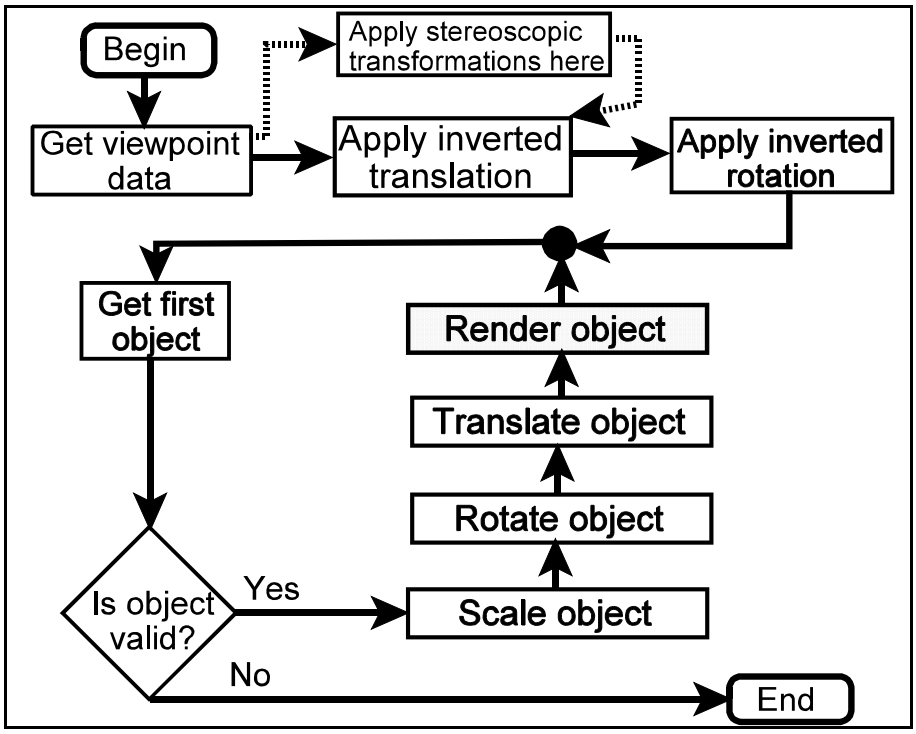


Figure 10: The algorithm used to construct the scene. Note that for stereo rendering, the scene is constructed twice and a translation is used to introduce the required disparity.

In addition to the above algorithm, two conditional blocks exist that construct the bounding spheres used in collision detection. The only difference between these blocks and the above algorithm is that they call the RenderBoundary method of class VisualRepresentation instead of the Render method. Note that for stereoscopic rendering, the scene is rendered twice and a top-level translation

introduces the binocular disparity required by shifting the final scene left or right, depending on which eye is being rendered to. This is what the overridden ThreadRoutine method of the VRStereoSink component does.

At this stage the image has been rendered the framebuffer but has not been displayed yet. Two commands, glFlush() and swapBuffers() force all pending drawing commands to be executed and swaps the colour buffers, effectively displaying the image. The modified glutMainLoop is then enabled with a call to VROutputDevice::setGlut and, finally, the rendering context is set to the default by calling VROutputDevice::setDefaultWindow.

Lastly, the whole ThreadRoutine is sandwiched between calls to VRSinkMonitor::startTimer and a VRSinkMonitor::ThreadRoutine call. The purpose of these two calls will be explained shortly.

4.3 The Virtual Reality Sink Monitor (VRSinkMonitor[1])

The most important task of the VRSinkMonitor component is to monitor the overall frame rate of the application and then do either

- nothing or
- report performance statistics at a regular interval or
- perform feedback optimisation

The first two options are arbitrary and will not be discussed at great length. The third option has already been introduced in detail in the background section and it now remains to be considered in terms of its implementation.

Before we start looking under the hood of the monitor, it is important to realise that this component is always active. When discussing the sink, I mentioned that the sink and VROutputDevice could be seen as an atomic unit. At this point I'd like to include the monitor in this unit. This means that the monitor is persistent throughout the lifetime of the sink (a bit like an agent[2]), whether its producing some form of output or not. The monitor was used extensively during the benchmarking

[1] Sometimes referred to simply as "the monitor"

[2] Persistence is one of the attributes used to define autonomous software agents. See [10]

phase of the project since, by instructing the it to gather performance results over a set period, we can gather accurate test data that can be subjected to further analysis.

So what happens when feedback optimisation is enabled? In this state, the monitor compares the average rendering period over a set number of frames to the user-specified performance index and then applies the algorithm illustrated in figure 8. The monitor treats the rendering context's capabilities as a stack: essential capabilities such as line and polygon rendering are at the bottom of the stack whereas expensive capabilities such as blending and texturing are assigned to the top of the stack. The monitor dynamically calls the VRCapabilities component to suspend or enable capabilities one at time and from the top of the stack downwards in order to keep performance above the specified frame rate. Thus, expensive and non-essential capabilities will be suspended first. This works well and is often sufficient to restore the frame rate to its desired levels.

The VRSinkMonitor component does not directly affect any part of the OpenGL pipeline, but if it is designed efficiently and implemented well, it could play a significant role in reducing bottlenecks at various stages in the pipeline. I found that in dynamic scenes the bottleneck often shifts back and forth between the geometric and rasterization stages of the pipeline. Thus, if the monitor were intelligent enough to identify where the bottleneck is currently overloading the pipeline and disable the capability that gives rise to the bottleneck, we could see substantial improvements in frame rate stability. Again I must stress that it is not the goal of feedback optimisation to improve overall performance - its sole aim is to ensure smooth, interactive animations by trying to maintain stable, real-time frame rates.

As stated before, choosing the sample period used to calculate the moving average is an important issue. Not only do we have to choose the sample period to minimise hysteresis, but also realise that our choice of sample period is platform dependent: on a SGI, a sample period of 15 frames works pretty well; however on a PC with no hardware acceleration, 15 frames is far too long - 5 frames is more realistic. I eventually chose a sample period of 5 frames for unaccelerated platforms and 15 frames for hardware accelerated platforms. These are empirical values that generally provided the best response, but for those applied mathematicians out there, any sample period around 15% of the maximum frame rate should produce a good response.

Now, let' s take a look at the methods that make up the VRSinkMonitor:

VRSinkMonitor: the constructor

The constructor is passed two pointers: one to the VRSink component that it will be monitoring and another to the VRCapabilities component. The constructor sets its internal pointers to these two components and then goes on to initialise other internal state variables, of which one is the sample period. Another data member worth mentioning is clock. This is an instantiation of class Timer and provides a mechanism for measuring rendering periods in microseconds. Two methods use the clock object:

startTimer

This initiates the clock object and sets the clock ticking, so to speak, by calling the Timer::mark() method.

getTime

This method calls the Timer::interval() method, which returns the time elapsed since the clock was started. This value is then returned by the getTime method.

report

This method takes two values as input parameters, the average frame rate of the last *n* frames as determined by the samplePeriod data member and the overall average frame rate. The method then displays these values to some output device. Currently, this device is a text console, but if more specialised output devices are used, it would be a good idea to override this method with one that will output directly to the output device. This will be important with head-mounted displays, where the user can see *only* the displays in the head unit.

SetPerformanceIndex

This sets the minimum frame rate to that specified by the user. This value will be used by the feedback optimisation mechanism when deciding whether to suspend or enable a capability.

getPerformanceIndex

This accessor method returns the minimum frame rate dictated by the user.

ThreadRoutine

The ThreadRoutine is called by the sink's ThreadRoutine once it has completed its rendering cycle and is about to return control to the main application program. It does the following tasks:
Firstly, it calls the clock to return the current time and calculates the rendering period of the last cycle. If that cycle constituted the last frame in a sample group, the sample average of the last *n*-1 cycles is calculated. The reason for discarding the data from the first cycle in each group is that, if feedback optimisation forced a capability to be enabled or suspended, then all display lists would have been recompiled during that cycle, thereby increasing the overall cycle period. Hence, that measurement would not be a true reflection of the time taken to render a frame and is therefore discarded.
This sample average, called the *moving average*, is then compared to the frame rate required by the user (referred to as the *performance index*). If the moving average is less than the performance index, the monitor calls the VRCapabilities::suspendCapability() method to suspend the highest active capability on the capability stack. If the moving average is greater than the performance index, the monitor enables a capability in the same way by calling the VRCapabilities::enableCapability() method.

When doing benchmarking, the ThreadRoutine also plays a crucial role by collecting sample data, storing it in an array and terminating the application when the test is complete. Before terminating the application, the collected data is written to a formatted data file that can later be imported directly into a spreadsheet and analysed. This is in fact how all of the tests were conducted: the test parameters were set, the test was run and the data file was then analysed using a spreadsheet. This was done to minimise errors introduced by manual sampling and calculation.

4.4 The Virtual Reality Capabilities Manager (VRCapabilities)

While doing background research, I came across Magician, a Java OpenGL API that also structures its whole system in terms of a set of interdependent component classes. One of these classes was the Capabilities class, an instance of which controlled and provided access to all the rendering capabilities available to the programmer. Experience proved this to be a simple, yet flexible class

and so I decided to do the same for the CorGi system: take all the scattered capabilities and aggregate them into an object capable of both handling capabilities and providing controlled access to them. Hence, the VRCapabilities component.

In order for the VRSinkMonitor to suspend and enable rendering capabilities, its important that these capabilities are all controlled by one global component. As we will see later, all the VisualRepresentation components in an application also rely on being able to access some global capability manager when parsing input files and rendering . Thus, the capabilities manager plays a small but vital role in coordinating capabilities in an application.

Another advantage of having a dedicated component to control capabilities is that it allows the user more explicit control at run-time of the current capabilities. This allows the programmer to suspend capabilities that aren't needed or to choose their own priorities to capabilities. For example, with the default feedback optimisation, one of the first capabilities suspended is texture mapping. If, however, the programmer decides that texture mapping is crucial, she can specify her own order of the capability stack or in fact implement a completely different feedback algorithm. This contributes tremendously to the flexibility of the system.

A capability can be in one of three states:

a. **Enabled**: the capability will be used when rendering the object
b. **Suspended**: the capability can be used but has been temporarily suspended by the application
c. **Disabled**: a data input error has occurred and the system is incapable of using the capability

A disabled condition arises when components are unable to use the capability due to lack of input data. This is typically caused by missing structure, colour or texture data files. Once a capability is disabled, it remains disabled for the entire duration of the program's execution.

There are two enumerations used by the VRCapabilities class: the first, Capability, enumerates all the capabilities:

```
enum Capability
 {
        LINES,
        POLYGONS,
```

```
        SMOOTH_SHADING,
        COLOUR,
        TEXTURES,
        ALPHA_BLENDING,
        SMOOTH_FILTERING
    }
```

The second, CapabilityState, enumerates the three states applicable to a capability:

```
    enum CapabilityState
    {
        DISABLED,
        SUSPENDED,
        ENABLED
    };
```

The only data member is the array of capability states:

```
CapabilityState Capabilities [NO_OF_CAPABILITIES]
```

where **NO_OF_CAPABILITIES** is some predefined constant. The reason for the Capability enumeration is to enable intuitive indexing to the Capabilities array. For example, if I want to disable texturing, I could use the following line of code:

```
Capabilities[(int)TEXTURES] = DISABLED;
```

Such a call is only valid within the VRCapabilities component, since the above array is declared protected, but accessor methods are provided that make operations on capabilities even simpler. Let's take a look at them:

VRCapabilities: the constructor

The constructor simply initialises all the elements of the Capabilities array to the **ENABLED** state.

toString

This method takes a parameter of type Capability and returns the string representation of that capability. For example the call

```
toString(BLENDING);
```

will return "BLENDING".

enableCapability(Capability cap)

This accessor sets the element of the Capabilities array denoted by the parameter cap to **ENABLED**.

suspendCapability(Capability cap)

This accessor sets the element of the Capabilities array denoted by the parameter cap to **SUSPENDED**.

disableCapability(Capability cap)

This accessor sets the element of the Capabilities array denoted by the parameter cap to **DISABLED**.

enableCapability(void)

This accessor sets the highest currently suspended element of the Capabilities array to **ENABLED**.

suspendCapability(void)

This accessor sets the highest currently enabled element of the Capabilities array to **SUSPENDED**.

getCapabilities(void)

This accessor returns a pointer to the Capabilities array.

4.5 The Visual Representation component (VisualRepresentation)

The VisualRepresentation component is the one component that represents an object in the virtual world. It contains all the data members and methods necessary to recreate from a set of input data files a complete visual representation of the object in the virtual world. Not only does this component know what the object it represents looks like, it also knows how to render it. In many respects, the visual representation is the core component of the system: all the rendering techniques are implemented within these methods and nearly all attempts at optimisation are found here. Put succinctly: if you want to see anything, you need this component. For these reasons, and because it is by far the most complex of all the components encountered in this thesis, I will discuss the VisualRepresentation class in detail.

The tasks of the VisualRepresentation can be separated into three stages:

i. Reading data from input files, parsing it and storing the results in the data members
ii. Compiling a display list using the data stored within the data members
iii. Rendering the object by executing the display list

Before I dive into a detailed explanation of the above steps, I think it would be wise to look at the data members, as they are complex in themselves.

The first data member class, **PolygonIndex**, is itself a class with only one data field: and array of integers, **Indexes**. The elements of this array serve as indexes into an array of vertex data. The reason for using look-up indexes is to avoid storing all the data for a single vertex more than once if that vertex is shared by multiple polygons. This situation arises quite often when using polygonal meshes. This class forms the elements of the array of polygons, or faces, that constitute the polygon mesh approximating the object. This array is called **ListOfFaces**. Essentially ListOfFaces is a two-dimensional array of $v \times n$ indices into a vertex array, where n is the number of polygons constituting the model or object and v is the number of vertices of each polygon. v was chosen to be 3 for three main reasons: firstly, by using triangles, we obey the simple convex polygon rule discussed earlier, since triangles are the only polygons guaranteed to have all three vertices lying in the same plane; secondly, the vast majority of models constructed today use triangular tessellation; thirdly, not only does OpenGL offer optimised drawing routines for triangulated models but many graphics devices offer hardware acceleration when processing triangles.

The array that actually stores the three-dimensional coordinates of all the vertices is called **ListOfVertices.** Each element of VertexNormals is of type **Vector3D**, a class that contains three data members for the x,y and z coordinates as well as providing operators for all the mathematical operations typically applied to three-dimensional vectors.

The integers **NumberOfFaces** and **NumberOfVertices** store the number of faces and vertices of the model, respectively.

Two other arrays containing vectors are used when calculating vertex normals: the first is called

FaceNormals and the second is called **VertexNormals**.

When it comes to storing colour data, there are three structures that need to be discussed. The first is the **VertexColour** record. This record stores a complete set of the following material:

- An array consisting of the RGBA ambient values
- An array consisting of the RGBA diffuse values
- An array consisting of the RGBA specular values
- The shininess constant

This record forms the structural element of the **ListOfColours** array. This is really the colour equivalent of the ListOfVertices array. The equivalent of array ListOfFaces is the **ColourData** array, which is also an array of indices used to look up a VertexColour record in the ListOfColours array. The integer variable NumberOfColours stores the number of colours used by this object and is equal to the number of records stored in the ListOfColours array. Lastly, there is the **COLOUR_STATUS** variable, which keeps track of the status of colour operations: if at any stage something goes wrong with inputting and parsing the colour data, COLOUR_STATUS is set to false.

Lastly, let's look at the texture mapping data structures. **VertexTexture** is the texture mapping equivalent of the VertexColour record and contains the following fields:

- **faceNo**, the number of the face (polygon) to which a texture is applied
- **textureNo**, the number of the texture that is being applied to the face
- **s1, t1**: the *s* and *t* texture coordinates of vertex 1 of the polygon
- **s2, t2**, the *s* and *t* texture coordinates of vertex 2 of the polygon
- **s3, t3**, the *s* and *t* texture coordinates of vertex 3 of the polygon

This record forms the structural element of the **TextureData** array and the variable **NumberOfTextures** stores the number of textures mapped to that object. Finally, **TEXTURE_STATUS** keeps track of the texture data input process and will be set to false should anything go wrong.

The **capabilities** array is a local copy of the VRCapabilites.Capabilities array and stores the capabilities to be used to draw this object. This is aimed at future work which switches capabilities on a per-object basis instead of all objects in the scene.

The **listName** variable stores a unique integer used to identify the display list that represents this object. The **ListOfTextures** array stores a list of unique integers used to identify all the texture objects used to texture map this object.
Two pointers exist to other classes used by the VisualRepresentation. The first, once initialised, points to a TextureManager component and the second points to BoundingSphere object.

The BoundingSphere object is instantiated from the BoundingSphere class, a simple class that contains the three-dimensional coordinates of the centre of a sphere surrounding the object (which we will render with the VisualRepresentation component) and the radius of this sphere. A default constructor initialises all these data members to 0.0.

Now, let's come back to the fundamental tasks of the VisualRepresentation and see how these are achieved using its methods. The first task, that of reading data from input files, parsing it and storing the results in data members, is performed by the **readOFF**, **readCOL** and **readTEX** methods. These are called by the constructor and are designed to be overridden if the data is to be obtained from sources other than files (such as streams). The task of compiling the display list that draws the object is performed by the **update** method and is called by the constructor once the data files have been read. Executing the display list is handled by the **Render** method and is called by the VRSink component as described previously. Let's look at each of these stages in more detail:

readOFF

This method reads structural data from .off files. The reason this format was chosen for the input data file is that the OFF format is simple to use: no special parser is needed to modify the data in the file - it can be edited directly using any text editor. It was also straightforward writing code that opens the file and reads in the number of vertices, edges and faces, followed by the three-dimensional coordinates of each vertex used. Once the vertices have been defined, a list of indices follows denoting which of the predefined vertices is used by the points of the triangles tessellating the object. The readOFF method stores the data read from the OFF file as follows:

- the number of vertices is stored in NumberOfVertices
- the number of faces is stored in NumberOfFaces
- the vertex coordinates are stored in ListOfVertices

- the indices used to access data in the ListOfVertices array is stored in ListOfFaces

The readOFF performs two other vital tasks: firstly, it constructs the bounding sphere used in collision detection and secondly, it calculates the normal vector for each vertex. Once the three-dimensional coordinates of each vertex has been read in, we have sufficient data to calculated normal vectors. The first step is to calculate the normal vectors of each triangle and normalise them as described in the Background section of this thesis. These normals are stored in the array **FaceNormals**. These vectors are then used to find the average normal at each vertex by calculating the normalised average of all the polygons sharing a
vertex. All vertex normals are then stored in the array **VertexNormals**.

readCOL

This method is reads colour data from .col files and stores it in the **NumberOfColours, ListOfColours** and **ColourData** data members. Like the .off files, the .col file is a text file that can be modified directly if required. It is deliberately constructed to be similar in structure to the .off file, since changes made to structure file often have to be reflected in the colour data file. This makes editing and object's data set a lot simpler. The data read in and the data members containing that data is as follows:

- the number of colours, stored in **NumberOfColours**
- the number of vertices. This is not used
- the full RGBA diffuse, ambient, specular and shininess values for each colour. Each colour is stored as a record in the **ListOfColours** array.
- next is the list of vertex numbers and the corresponding colour index for each vertex.

As will see when discussing the update method, a vertex is assigned a default colour if no colour data is found in the data file for that vertex.

readTEX

This method is responsible for reading all the data required to map textures onto this object. It also creates the texture objects, reads in the texture coordinates and sets up the texture mapping data members. Although the .tex data file is again very similar in format to the .off and .col files, there are some key differences. The header contains the number of textures and the number of *faces* (not vertices) that are texture mapped. This is followed by a list of filenames of the textures used, which

is then followed by the texture coordinate block. Here, each line contains the face number, the texture number of the texture mapped to that face and the six texture coordinates needed to map the face.

The method starts off by reading and storing the number of textures and number of faces to the **NumberOfTextures** and **faceNumber** data members, respectively. It then uses **NumberOfTextures** to allocate enough space in the **ListOfTextures** array to store all the unique integers that will be used to reference the texture objects. It also uses **NumberOfFaces** to allocate enough space to **TextureData** to store the texture coordinates for all the faces of the model that are texture mapped. Once this is done, the method reads in the texture map filenames and for each one calls the **TextureManager::createTexture** method, which creates the texture object and its associated texture(s) and returns the unique unsigned integer that identifies that texture object. These texture object identifiers are stored in the **ListOfTextures** array. Once the texture objects are created successfully, the method goes on to the texture coordinates block. For each line, it stores the face number in the **FaceNo** field of the **VertexTexture** record for that face, the texture number in the **textureNo** field and the texture coordinates in the remaining fields. If any of the above stages fails, the **TEXTURE_STATUS** flag variable is set to false, disabling texture mapping.

update

The update method compiles all the commands necessary to render the object in three-dimensional space into a platform independent set of commands, the display list. While constructing the display list, OpenGL performs numerous optimisations. In fact, many operations (such as transformations) will only have their final results stored in the list, ensuring that rendering via the execution of display lists is at least as fast as *immediate mode*, or direct execution. While compiling the list, the update method has to interrogate the local capabilities status array to see which capabilities to compile into the list and which to forego. An important point I realised when rewriting the update method to support feedback optimisation is that it is not good enough simply to disable OpenGL's processing flag for a capability by using the glDisable() command. Most of the capabilities will rely on many OpenGL commands during the execution of the update method to changes the processing state of the pipeline and thus affect the final outcome of the scene. Tests showed that if these commands were not taken into consideration, the change in rendering performance proved negligible. It would also reflect bad design to have compiled display lists containing commands that only hinder

rendering without contributing at all to the final scene. As a result, the update method is full of checks into the local capabilities array. You might be tempted to think this is a Bad Thing, but realise that these checks are only executed when the display list is first compiled or recompiled due to a feedback optimisation. In addition, the display list stores only the final OpenGL commands, so these capability checks are not included in the set of commands that are eventually executed.

So what does the update method actually do when compiling the list? Well, that's the subject of the next couple of paragraphs:
The first thing the method does is interrogate the **COLOUR_STATUS** and **TEXTURE_STATUS** flags to see if the colour and texture data was read in successfully. If not, the elements in the capabilities array corresponding to these two states are set to **DISABLED**.

The next thing the update method does is to see if the texture objects used by the texture mapping routines exist within the current rendering context. If not, a call is made to the **TextureManager::rebindTexture** method to remedy the situation.

One of the exciting prospects I looked forward to was using vertex arrays to optimise the rendering process. However, the OpenGL vertex-, colour-, normal- and texture coordinate arrays all expect flat linear arrays as input parameters to the glVertexPointer, glColourPointer, glNormalPointer and glTexCoordPointer commands. So the next thing the update method does is extract this data from the complex data member structures and stores it in linear arrays, which are then used as input to the above commands. Once this step is complete, our data is ready and we are set to actually start compiling the display list.

The glNewList command indicates the beginning of the display list[1] and the first thing the method does is set the material properties to a default of white. This is done to counteract the ominous black screen that could arise if colour processing was disabled due to input errors. Then follows a block of commands that interrogates the capabilities array and prepares the OpenGL capabilities that will

[1] *A Word Of Wisdom*: the start and end of a display list are denoted by the glNewList and glEndList commands, respectively and the programmer has to take care, since not all OpenGL commands are allowed between these two commands. If you use display lists and start getting weird results (or "undefined behaviour", as the *OpenGL Programming Guide* puts it), look for commands that don't belong between the glNewList and glEndList pair.

be used. Once this stage is complete, we are ready to start rendering the polygons and call the glBegin command with the GL_TRIANGLES constant.

To actually render the polygons, we iterate through the data arrays and call OpenGL commands that specify the vertex coordinates, normals, colours and texture coordinates of each polygon that forms a facet of the polygon mesh approximating our object. This can be done in one of two ways. If we use vertex arrays, we need only call glArrayElement to extract and render all the data pertaining to one vertex; glDrawElements does the same, but processes whole polygons at a time; glDrawArrays is used to extract the needed data and render the entire mesh all at once. This is convenient and potentially very efficient but lacks flexibility. For instance, we cannot change the current texture being mapped prior to a polygon being rendered if we use glDrawArrays to render the model. Thus, when using vertex arrays, I render one triangle at a time using the glDrawElements command. If we don't use vertex arrays, we explicitly have to call glVertex** command to specify the vertex coordinates, the glNormal**[1] command to specify the vertex normal coordinates, the glColor** command to specify the RGBA colour and the glTexCoord** command to specify the texture coordinates associated with each vertex of every polygon, for all polygons. I trust the reader can appreciate how expensive rendering becomes if for each vertex in a complex model, we have to call four OpenGL commands. If, in addition, we modify the material properties (an expensive operation) of each vertex (or even polygon), performance rates plummet significantly. For this reason, I chose to set the material properties once for each object.

Prior to processing each polygon, the glBindTexture command is called to activate the texture object needed to texture map that polygon. This is of course subject to the texture mapping capability being enabled.

Once all the polygons have been processed, we call glEnd to specify the end of the primitive rendering block, call glEndList to specify the end of the display list and finally, disable the OpenGL processing of capabilities.

---

[1] For many of its commands, OpenGL has suffixes that denote the type and number of parameters passed to that command. I will use a * to take the place of such a suffix. ** would then indicate that the command takes two suffix characters.

Render

The VisualRepresentation::Render method is responsible for executing the commands stored in the display list and actually rendering the object. But first there are a couple of status checks it must perform. Firstly, it compares the current **capabilities** array with the global **Capabilities** array to determine if the set of active capabilities has changed since the last time the object was rendered. Secondly, it checks that the display list that will be executed is valid. If either the active capabilities have changed or the list is invalid, the update method is called to recompile the list. Otherwise, the display list is executed using the glCallList command and the object is drawn to the framebuffer.

setTextureManager

This method accepts a pointer to the global TextureManager component and assigns it to the local TextureManager pointer.

GetBoundingSphere

This method creates a new bounding sphere object and returns a pointer to it.

RenderBoundary

This method draws a transparent sphere according to the parameters contained in the BoundingSphere object, as returned by a call to GetBoundingSphere.

4.6 The Texture Manager Component (TextureManager)

Because there is such a wide range of options available when mapping textures to an object's polygons, texture mapping is a complicated process. Furthermore, applying textures is by far the most computationally intensive feature of the rendering system, so implementing it needed special consideration in terms of any optimisations that could be applied. For these reasons, I chose to create the TextureManager, a component that acts as a global texture resource manager of all the textures that exist within the scope of the current rendering context. This would avoid the situation where we have multiple objects in the scene using the same textures and creating duplicate texture objects. If all textures are created and monitored globally, the VisualRepresentation components can share texture objects, thereby removing redundancy and hopefully improving performance. The TextureManager is implemented as a linked list of TextureListNode objects and three methods that

create texture objects, load the texture maps and rebinds the texture objects, if necessary.

The TextureListNode class contains constructors and two data members: **textureID**, the unique unsigned integer that identifies this texture object and **fileName**, the name of the file containing the actual texture map.

TextureListNode : the constructors

The default constructor sets **textureID** to 0, which denotes the default texture (i.e. no texture) and assigns an empty string to **fileName**. The second constructor takes as its only parameter a string containing the texture map file name. It then calls glGenLists, an OpenGL command that chooses the unique texture identifier, and assigns the result to **textureID**. **fileName** is assigned the value of the parameter.

TextureManager : the constructor

The constructor's only task is to initialise the linked list. To do this, it instantiates a linkedlist object from the linkedlist template class, using the TextureListNode class as a template for the objects that constitute the elements of the list.

createTexture

This method is called by the VisualRepresentation to create a texture object for each texture required to texture map that object. The method first scans through the elements of the linked list to determine in the texture object already exists. If it does, the **textureID** of that texture object is returned to the calling VisualRepresentation component. Otherwise, it creates a new TextureListNode and, if the node is created successfully, calls the glBindTexture command to generate the texture object. It then sets all the texture mapping parameters to the following values:

- Minification filtering: GL_LINEAR. This uses linear interpolation of the four closest pixels to approximate the pixel in the reduced texture map
- Magnification filtering: GL_LINEAR. Again, linear interpolation is used to approximate the new pixel
- Horizontal texture wrapping: GL_REPEAT. This specified that, if a texture does not fill a polygon, it is repeated, or tiled horizontally. The parameter for vertical texture wrapping is set to the same value.

- Current texturing function: GL_MODULATE. This specifies that the colour of the texture map should modulate the original colour of the surface without texture mapping.

Once these parameters are set, **createTexture** calls the **loadTexture** method, discussed next, to load the texture data from the file passed by the VisualRepresentation component. After this is done, the **Add** method of the linked list is called to add the TextureListNode representing this texture object to the end of the list. The current texture is then set to the default texture (no texture) so that no objects processed after the current one have the same texture applied to them too.

loadTexture

This method extracts texture data contained in gif (graphic interchange format) files and eventually passes it to the texture object responsible for that texture. It does this by first using the **giftopnm** system utility to convert the data in gif to RGB format and then using another system utility, **pnmflip,** to reverse the order of the data to be compatible with data alignment of the OpenGL texture objects. This portable graphic file is then opened and the header scanned to get the dimensions of the texture map. The method then allocates sufficient memory to store the data using data extracted from the header and the malloc command. Once this data is successfully read in, we pass it to the gluBuild2DMipMaps. This utility produces the set of prefiltered mipmaps of varying resolutions and calls the glTexImage2D command with the correct parameters to store the set of mipmaps within the current texture object. Finally, it closes the data file and frees the memory allocated by the malloc command.

rebindTexture

The sole purpose of this method is to initialise texture objects that exist within different rendering contexts to the one wherein the initial texture processing took place. It takes a textureID as input parameter and calls glBindTexture to create a texture object. It then sets the texture mapping parameters discussed above and finally calls the loadTexture method to load the texture data.

5

Benchmarking Results

The following section outlines and the presents the results of four tests designed to quantify the performance of the rendering system. These results were then analysed to identify possible weaknesses in the system. The following tests were performed:

- Test 1: ***Investigating rendering performance with hardware acceleration with regards to the number of objects rendered and the number of active capabilities***. This test was aimed at getting some idea of the performance curves the system is subject to on hardware accelerated platforms by varying both the complexity of the scene (the number of objects) and the capabilities used to render the scene.
- Test 2: ***Investigating rendering performance with software acceleration with regards to the number of objects rendered and the number of active capabilities***. This test was also aimed at quantifying the performance curves, but this time without the benefit of hardware acceleration. Again, the varying parameters of the test were the number of objects and the active capabilities used to render the scene.
- Test 3: ***Investigating remote rendering performance with hardware acceleration with regards to the number of objects rendered and the number of active capabilities***. Again, the nature of the test remained the same, except that this time the performance of the client-server architecture was tested across a network by running the client on one machine and the server on another. Both machines supported hardware acceleration.
- Test 4: ***Investigating the efficiency and response of feedback optimisation in dynamic animations.*** This test was aimed at determining whether feedback optimisation did in fact ensure that performance levels stayed above a specified frame rate and, just as importantly, what performance impact it had on the system.

A special benchmarking application was written that rendered a standard scene containing a specified number of objects. The scene was constructed so that no object obstructed another (to minimise hidden surface removal) and ensured that no object left the scene at any time. The benchmarking program is passed two parameters: the number of objects to render and the required

minimum frame rate, if any.

Depending on the nature of the test, slight modifications were made to the VRSinkMonitor component that allowed it to write its monitoring data to a file. Shell scripts were used to run sequential benchmarking programs for a different number of objects in the scene.

A maximum of 100 objects could be fitted on the screen. The number of objects used in the tests were 1, 2, 4, 8, 16, 32, 64 and 100. This exponential distribution was chosen to highlight interesting behaviour that the system exhibited with between 1 and 20 objects in the scene.

Double-buffering was disabled for all tests, since this introduced delays due to the refresh cycles that produced misleading frame rate results. With double-buffering enabled, the maximum frame rate is usually the refresh rate of the monitor. See [12] for more details on benchmarking.

When looking at the graphs, the reader will see that a set of numbers denote the active capabilities These numbers correspond to the following:

	Lines	**Polygons**	**Smooth Shading**	**Colour**	**Textures**	**Blending**	**Smooth Texture Filtering**
7	Yes	Yes	Yes	Yes	Yes	Yes	Yes
6	Yes	Yes	Yes	Yes	Yes	Yes	No
5	Yes	Yes	Yes	Yes	Yes	No	No
4	Yes	Yes	Yes	Yes	No	No	No
3	Yes	Yes	Yes	No	No	No	No
2	Yes	Yes	No	No	No	No	No
1	Yes	No	No	No	No	No	No

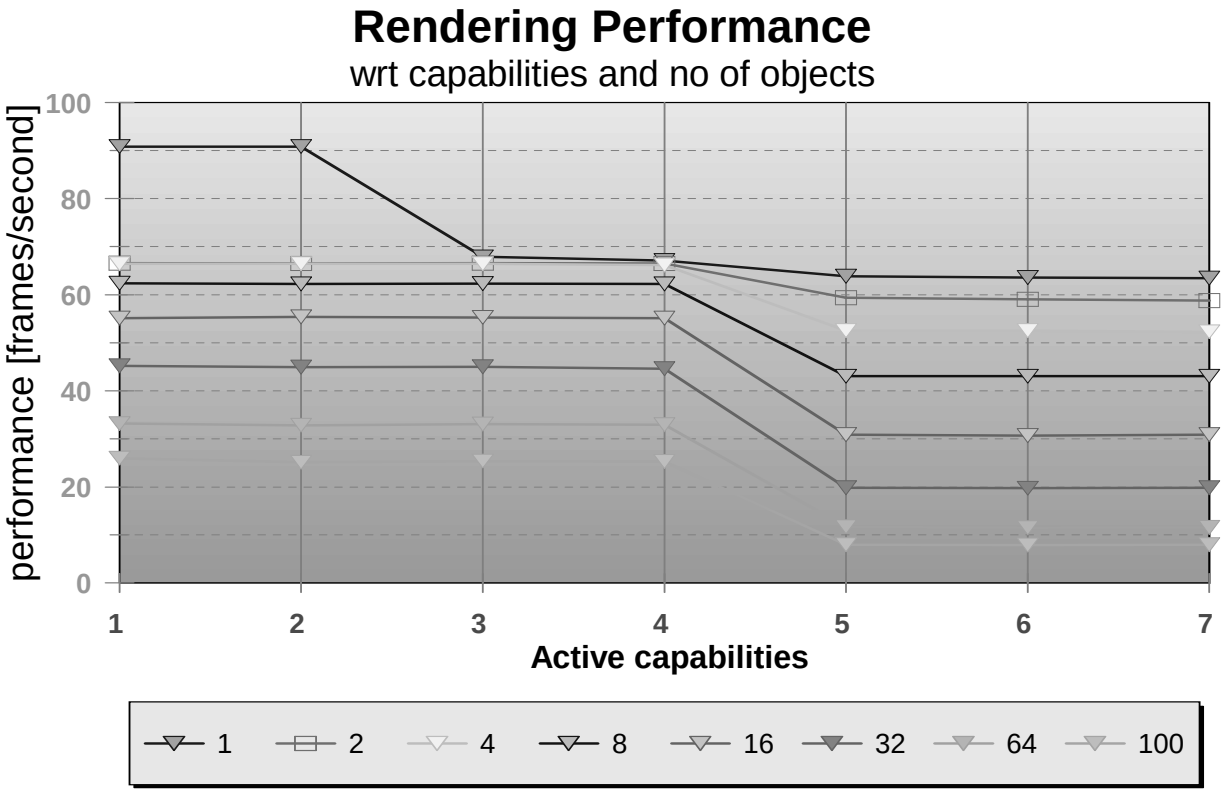
5.1 Test 1:

Investigating rendering performance with hardware acceleration with regards to the number of objects rendered and the number of active capabilities.

Description

This test was performed on a SGI Octane with hardware acceleration supported. The VRSinkMonitor component was slightly modified to render 150 frames with the current set of active capabilities, suspend the capability at the top of the capability stack, render the next 150 frames, suspend the next capability, and so on until the capability stack is empty. This takes place with a fixed number of objects in the scene. To gather data for different number of objects in the scene, a Unix shell script executes the above test with different parameters: 1, 2, 4, 8, 16, 32, 64 and 100.

The results were as follows:



Discussion

The flat curves are characteristic of hardware accelerated platforms and illustrate how adding further quality to the scene by enabling capabilities has no impact on rendering performance. The notable exception is texture mapping. Even with the extensive texture mapping resources supported by the Octane, enabling texture mapping caused an average drop of 37% in performance.

5.2 Test 2:

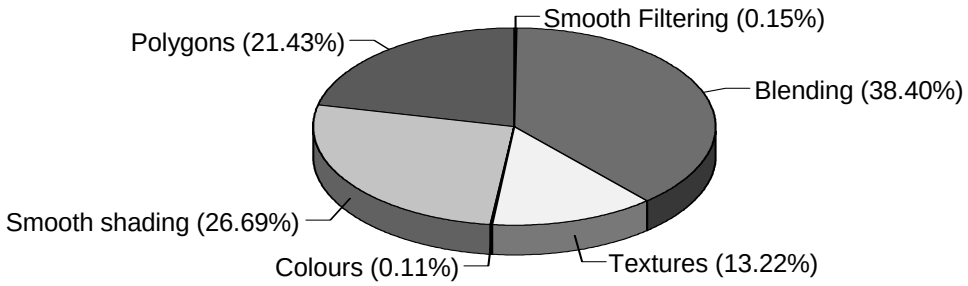
Investigating rendering performance with software acceleration with regards to the number of objects rendered and the number of active capabilities.

Description

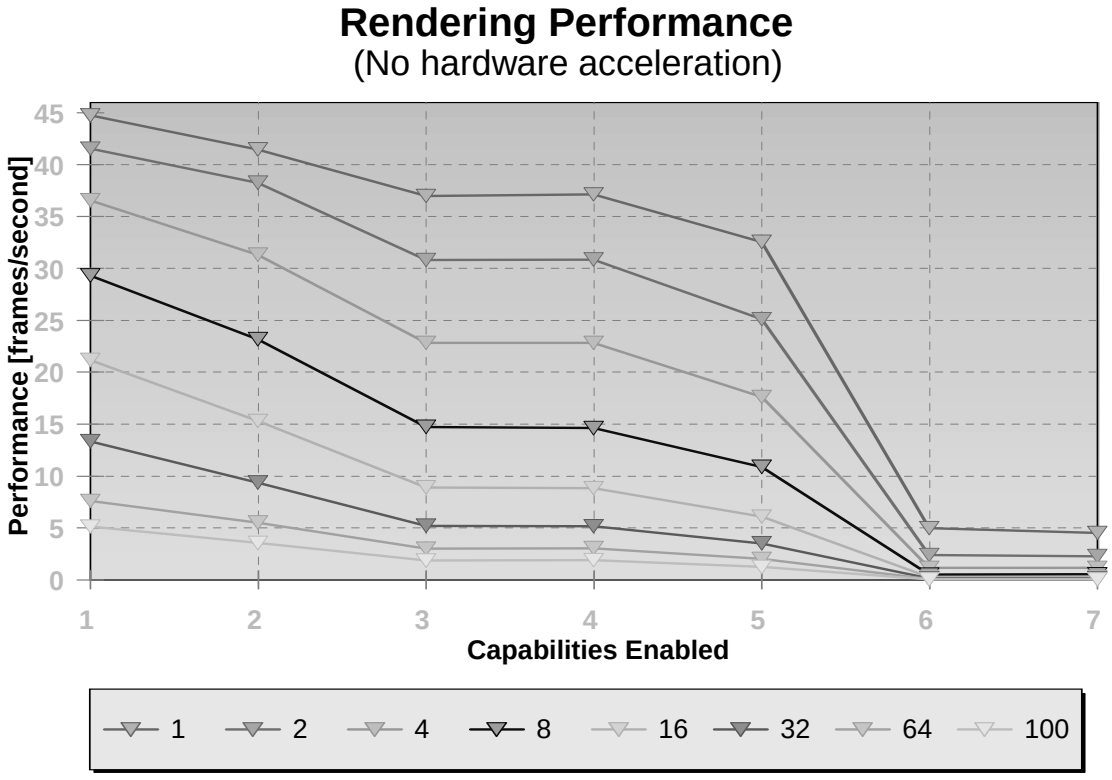
This test was performed on an SGI Octane. The benchmarking application was compiled using the Mesa graphics library, an OpenGL clone that only uses software acceleration. The same VRSinkMonitor component was used as with test 1.

Discussion

As you can see, the rendering performance curves change significantly when we disable hardware acceleration and leave the CPU to do all the rendering. The impact of each capability as it is enabled is also much more evident. One fact that is undeniable is the crucial role that hardware acceleration plays: hardware accelerated rendering is at least twice as fast as software based rendering and much more stable across the range of capabilities. The following chart illustrates the average performance penalties associated with each capability using software based rendering:



Although texture mapping still requires a good portion of the resources, it is the computationally intensive capabilities, notably blending and smooth shading, that make the most impression on the performance levels. In fact, these two capabilities combined consume almost two thirds of the rendering resources. Note also the significant penalty we pay for drawing polygons without acceleration.

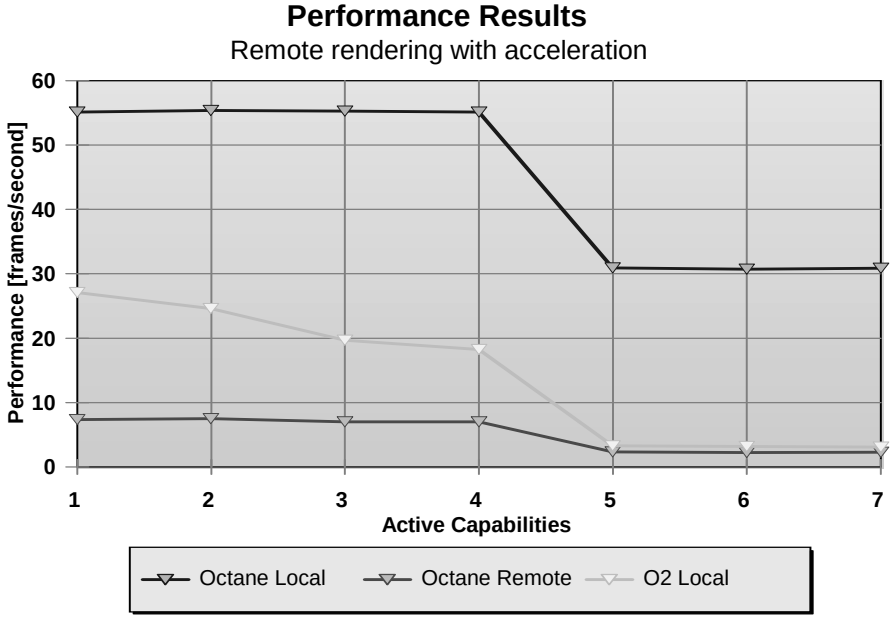


5.3 Test 3:

Investigating remote rendering performance with hardware acceleration with regards to the number of objects rendered and the number of active capabilities

Description

This test is identical in nature to the two previous test, except that it was aimed at testing the performance of rendering across a network using OpenGL's client-server architecture. The SGI used in the previous tests executed the application, but the test was rendered on a SGI O_2 connected to the same segment of the network. The results were as follows:



Although this test was performed with varying numbers of objects in the scene, I chose to compare the remote rendering performance to the local rendering performance of both the Octane and the O_2. All three curves represent the performance levels with 16 objects in the scene (1008 polygons).

As you can see the two local rendering curves are very similar, except for the obvious difference in levels of performance. The red curve, however, is the one of interest. It shows the impact of rendering across a network. It's hard to qualify what factors actually contribute to these final figures, since the client-server aspect of OpenGL is poorly documented. I suspect network bandwidth plays

an important role and that transport delays might damp out any performance gains due to suspended capabilities.

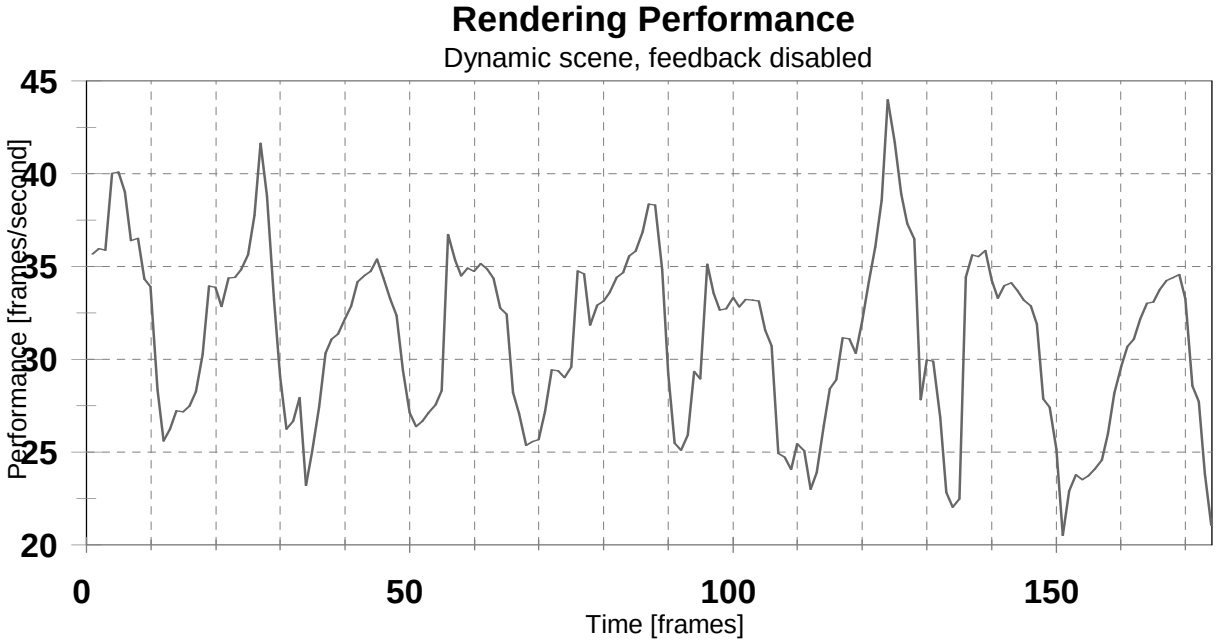
5.4 Test 4:

Investigating the efficiency and response of feedback optimisation in dynamic animations.

Description

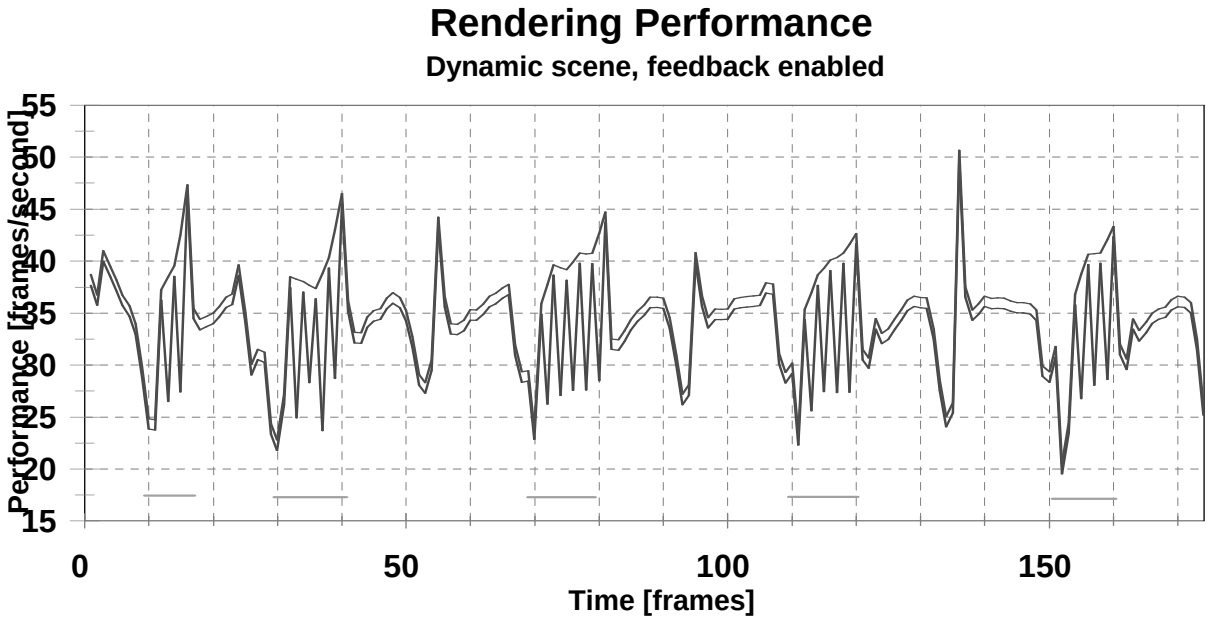
In order to get some idea of the actual behaviour of the feedback optimisation mechanism, I put it to work in a highly dynamic scene. This scene contained ten animated objects rendered with full capabilities. Furthermore, these objects periodically left and re-entered the scene, very often right in front of the viewpoint. In many ways, it is a worst case scenario for the feedback mechanism.

I have included two graphs: the first with feedback optimisation disabled to give the reader an idea of the rendering performance as a function of time. The second graph contains two curves: the actual performance with feedback optimisation set to optimise performance at 30 frames per second (the



red curve) and another curve (green) which will be discussed shortly.

As you can see, frame rates are far from constant in dynamic scenes. This is due to different parts of the pipeline being overloaded at different times. For instance, the peaks we see at frame 34 and 124 are due to large texture maps suddenly disappearing from the scene, thus relieving both the geometric and rasterization parts of the pipeline. In contrast, the low frame rate at frame 151 is due to a high quality texture mapped object moving into a position right in front of the viewpoint. This firstly requires huge amounts of texture filtering and secondly, almost all of the pixels in the window to be rendered, straining both the geometric and rasterization stages of the pipeline.



The areas to focus on in this graph are the areas above the purple bars. These are the period during which feedback optimisation is actually active. The red curve representing the optimised performance oscillates significantly during these periods. The reason? Texture mapping. This capability makes such an impact on the rendering performance that we can usually restore frame rates to well above the required frame rate simply by suspending texture mapping and this is what the feedback mechanism does. The problem arises due to the fact that the feedback mechanism finds the frame rate to be well above the required rate and re-enables texture mapping. Performance plummets and so texture mapping is suspended, and so the cycle is repeated until the performance,

with texture mapping enabled is greater than the performance index. Since the display lists of all objects are recompiled when a capability state change occurs, it is crucial that we minimises these state changes. The second curve (green) illustrates how an improved algorithm achieves this by detecting when, by suspending texture mapping, we cause a period of flux and reacting by completely suspending texture mapping until frame rates are sufficiently high to safely re-enable it.

6

Problems encountered

Vertex arrays

The first problem to hamper the success of this project was that vertex arrays refused to work on when rendering remotely. As you've seen, the performance of remote rendering is rather poor and I had hoped to see vertex arrays showing the way to better performance. The problem with vertex arrays is that it is built upon the client-server architecture of OpenGL and (as I have commented on numerous occasions) this is a poorly documented part of OpenGL. Is this because nobody uses it?

Stereo output devices

The second problem is shortage of output devices required to render the dual images required to implement true stereoscopic rendering. This was one of the biggest disappointments of the year, since one of the hallmarks of virtual reality systems is immersive, three-dimensional graphics. Due to this lack of hardware, I was not able to test the VRStereoSink component.

OpenGL on Win32 platform

Another big disappointment is both the bad performance and implementation of the Windows 9x and NT OpenGL binding (In the immortal words of somebody I know: "Bad, bad Windows..."). This API, unlike its SGI counterpart, is quite happy to crash a compiler and/or debugger and also refuses to calculate correct perspective projection coordinates. Fortunately, SGI is working with Microsoft on a new binding that promises greater stability and accuracy.

Bugs

Turning to the code, there are two major problems that were encountered. The first is a memory leak caused either by the texture manager or other texture mapping routines. This is hampering the overall performance of the system. If it could be removed, I think we could see a significant improvement in performance, especially in complex scenes.

Another mysterious bug that I came across while trying to perform tests on a PC caused the application to crash completely when more than 16 objects existed in the scene and reported a bug in file hash.c Unfortunately, this prevented me from doing valuable tests of the system on a Linux PC platform in addition to the SGI Iris platform.

7

Conclusion

This project successfully implemented a set of flexible, efficient and platform independent components for rendering real-time virtual realty graphics. The object oriented design of these components allows specialised components to be developed easily using inheritance and polymorphism. The project succeeded in improving upon the performance of the RhoVer system. The performance results of various rendering tests were presented and from this the following conclusions were drawn. Firstly, it was concluded that hardware graphics acceleration is crucial for successfully implementing interactive rendering. Secondly, texture mapping was identified as a major target for future optimisation. It was also concluded that the current feedback optimisation algorithm could be improved and suggestions were made as to how this could be achieved. Problems encountered included vertex array based rendering, lack of stereo hardware, poor OpenGL Win32 implementations and unresolved software errors.

8

Future Work And Possible Extensions

The most immediate extension would be to implement a feedback algorithm that switches capabilities on a per object basis, theoretically resulting in smoother optimisation curves. This would require some VisualRepresentation status field for each object so that less important VisualRepresentations have their capabilities dropped first in order to improve the overall frame rate. This of course requires being able to control capabilities on a per object basis. But it’s well worth it. In this way, we use some of the free CPU cycles to alleviate bottlenecks in the graphics pipeline.

Secondly, we need to implement shadows as another capability. Shadows can play a crucial role in providing positional information about objects is a scene. Yet, shadows are expensive and most shadow algorithms requires the scene to be drawn at least twice[1]. Very recently, I came across a shadow generation library written by Mark Kilgard that uses multiple processors (if present - as is the case with the SGI Octane) to generate realistic soft shadows for real-time virtual reality systems. I'm curious to see if we could use this library to generate shadows and impact they will have on rendering performance.

Lastly, I think a good investigation into the OpenGL client-server architecture could prove very advantageous, since this could lead to the solution of the vertex array problem as well as improving remote rendering performance.

9

References

1. Woo, M; Neider, J; Davis, T: ***OpenGL Programming Guide, Second Edition***, Appendix F
2. Woo, M; Neider, J; Davis, T: ***OpenGL Programming Guide, Second Edition***, Appendix E
3. Woo, M; Neider, J; Davis, T: ***OpenGL Programming Guide, Second Edition***, Appendix F
4. Lin, F *et al.*: ***Geometry for Computer Graphics (Student Notes)***; Computer Graphics Unit, Manchester Computing Centre, University of Manchester.
5. Lilley, C *et al.*: ***Colour in Computer Graphics***; Computer Graphics Unit, Manchester Computing Centre, University of Manchester.
6. Woo, M; Neider, J; Davis, T: ***OpenGL Programming Guide, Second Edition***, Appendix E, pg 592.
7. Watt, A&M: ***Advanced Animation and Rendering Techniques: Theory and Practice***, ACM Press, 1992; pp 22-26
8. Foley, J ; Van Dam, A: ***Fundamentals of Interactive Computer Graphics***, Addison Wesley, 1984; pp 546-548
9. Franklin, S and Graesser, A: ***Is it an Agent, or just a Program?: A Taxonomy for***

[1] See [11] for a detailed discussion on shadow algorithms

Autonomous Agents; Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

10. Iris Insight Library, ***OpenGL on Silicon Graphics Systems***, Chapters 12 and 13
11. Watt, A&M: ***Advanced Animation and Rendering Techniques: Theory and Practice***, ACM Press, 1992; Chapter 5

10
Acknowledgements

I'd like to thank the following people for their invaluable contributions throughout the year:

Shaun Bangay, my supervisor, for often stopping me from going off on a wild goose chase and for always being helpful in keeping the bug population in my code to reasonable levels.

Mike Rourke and the other Masters students for their insights and help, both in presenting this project and throughout the year.

And last but not least, an affectionate thank-you to my accomplices, Romeo and Benvolio, for bearing the brunt of all my experiments and patiently handling all those rogue pointers...

Index