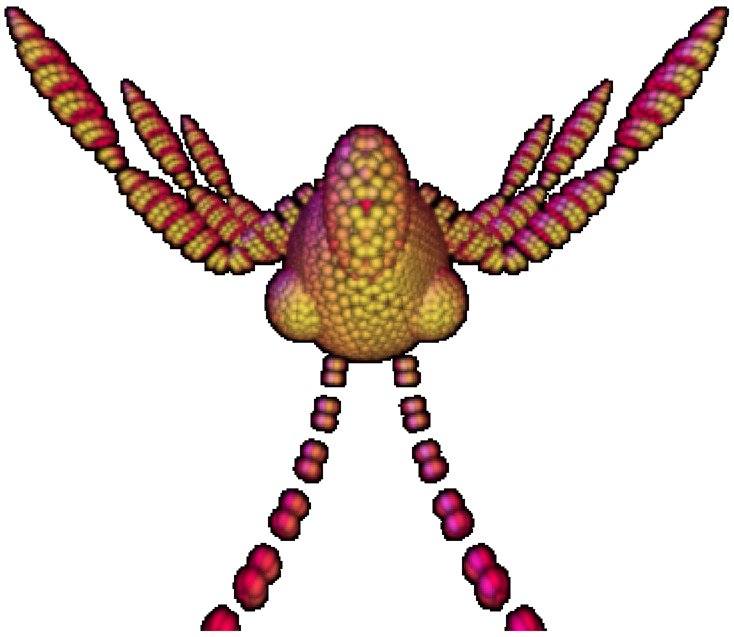
Exploring Rendering Techniques for a Virtual Holodeck

Chantelle Morkel (theriaca@lycos.com)
Computer Science Honours 2002



Submitted in partial fulfilment of the requirements for the degree of Bachelor of Science (Honours) of Rhodes University.

Abstract

The main objective of this research is to implement a “Star Trek”-like holodeck in a computer environment. An experiment to create graphical primitives and images solely out of spheres is being conducted. We investigate several approaches to creating primitives using spheres. Images are made up of the primitives within the system. Phong shading is adapted to work with spheres as the basic component (as opposed to pixels) of images. Results of this experiment are presented and discussed. We conclude that using spheres to create primitives and images is a viable approach to creating realistic-looking three-dimensional (3D) images.

Contents

Chapter 1

Michael Halle [Hal, 97] states that there is currently a renewed interest in three-dimensional imaging. It is to such a degree that it is now both popular and practical. Standard display technology is mainly two-dimensional (2D), the 3D equivalent is still being developed, but the prototypes are not within the means of individuals.

Through the use of depth cues, shading, texture mapping and more it is possible to display 3D images on a 2D display. [How, 02] Although not ideal, these techniques provide users with realistic looking images. While not essential, it would be nice to be able to interact with the 3D images (besides with the use of a keyboard and mouse).

Enter the holodeck. The idea behind the holodeck is to immerse a user in a 'virtual' world, made up of holographic images and 'solid' images (refer to section 2.1.2), and allow them to interact with their surroundings.

While neither the holographic hardware nor the 3D display to create the holodeck exists, we can explore implications for 3D rendering algorithms by applying them to a simulated, hypothetical technology. To create a suitable 'holographic technology', it is necessary to explore what technology is already in existence, and what technology is currently being developed.

The experiment to be conducted has to be both feasible and innovative. The resultant experiment creates images solely out of spheres. We assume the ability to holographically render spheres of a desired size and colour at any point within the holodeck. The holodeck is confined to an area of space, delimited by a cube.

1.1 Background Theory

The holodeck is completely fictional; it is a brainchild of Gene Roddenberry, creator of the popular science-fiction series "Star Trek" [Ale, 98]. The holographic component of the holodeck, however, is not fictional. The technology to project holograms exists, even if it is under-developed.

This section serves to briefly introduce the concept of holography.

1.1.1 Holography

Dennis Gabor, a Hungarian-born physicist, devised the theory of holography, because of this; Gabor is considered to be the ‘father’ of holography. In 1947, “Gabor proposed to employ a divergent electron beam, propogating beyond the focus of an electron microscope, to illuminate an object placed in the path of the beam and to record the result of the interaction between the electrons and the object at a distant detector” [Out et al., 99].

According to Outwater & Hamersveld [Out et al., 99], Gabor went on to prove his theory, not with an electron beam, but rather with a light beam from mercury vapour lamp squeezed through a pinhole and then colour filtering it. The result was the first hologram ever made.

1.2 Overview of this Thesis

Chapter 1 serves as an introduction to both the problem and the solution being implemented. Chapter 2 describes and evaluates work in the fields of 3D imaging and the holodeck. Chapter 3 describes the primitive rendering algorithms, as well as general system design. Chapter 4 discusses the implementation of shading, and the graphical user interfaces in the system. Chapter 5 presents results of experiments. Chapter 6 lists the conclusions reached.

Chapter 2

The first section, section 2.1 contains works that are directly related to the holodeck, they are either part of the definition of a holodeck, or outline work that has been done while attempting to create an actual holodeck. Section 2.2 contains work that defines the theory of holography, as well as what the current state of the art is with regards to holograph projection. The final section, section 2.3 makes mention of the graphics pipeline and how images are rendered, as well as how graphics hardware works.

2.1 The Holodeck

2.1.1 Concept

As mentioned, the idea behind the holodeck is to have holograms, with which one may interact, to create a virtual scene in which the user will be fully immersed. The goal of the holodeck then becomes realism. A user must experience the virtual world as though it really exists.

2.1.2 The Theory

The Star Trek holodeck uses five levels of simulation. [Bel, 94]

- Objects in the distance are holograms projected onto the walls
- Close objects are holograms that are projected into space
- Fixed objects are a combination of projected holograms and shaped forcebeams
- A static object is replicated onto the holodeck
- An animated object is comprised of ‘holomatter’ created by the transporter-replicators for use in the holodeck.

Considering that the technology does not actually exist, the only simulation, from the above list, that is applicable to our experiment is the projection of images into space.

2.1.3 State of the Art

2.1.3.1 Holodeck Approximations

This section serves to outline some of the approaches companies and/or individuals are taking to create their own version of the holodeck. Most of the works cited in this section are relevant to our project only in the sense that they are physically working systems, while our system is a computer simulation.

Fractal Shape Changing Robots

The Robodyne Cybernetics group has put forward a proposal to create a holodeck. Their intepretation of the holodeck is a combination of "Fractal Shape Changing Robots" (hardware reality) and headgear (virtual reality) [Mic, 97].

The shape changing robots and virtual reality helmet allow for the creation of simple objects such as chairs, tables and beds. The theory goes that each time a user gets close to an object that could be physically interacted with, the shape changing robots (Robots) will approximate the object to allow the user to interact with it. The system will only synthesise objects that are close by, while those that are further away exist in the VR helmet only. [Mic, 97]

Robodyne claims that their technology will be both commercialised and in general use in less than five years [Mic, 97]. It is now five years and more later, but none of the technology they outlined in their proposal exists.

The problem with the Robodyne proposal is that they have not stated clearly what they are going to do, but rather what they want to do. Their system is based on technology that exists only in primitive form, the area of application for nano-technology is small, and is still being researched and developed.

Temporarily ignoring the problems that arise out of the Robodyne system proposal, some of the approaches they suggest are worth further discussion. If reference is made back to section 2.1.2, it can be seen that Robodynes' approach to creating objects is the same as that of the holodeck. They propose to only create objects that are in the immediate vicinity of the user, while everything else is only a simulation. As a user approaches an object it becomes 'real'. To some degree this could be applied to the holographic technology we propose to create. When an image is rendered, depth testing may be used to hide the pieces of the image that the user cannot currently see anyway. This saves the processing for what is in front of the user.

The CyberSphere

Vinesh Raja and Julian Eyre, employees of the Warwick Manufacturing Group, have created the "Cybersphere" [Dun, 99].

The CyberSphere is a large, hollow translucent sphere. The user is placed inside the sphere, while images are projected onto the sphere surface. This allows user immersion into the virtual environment.

To facilitate movement, the CyberSphere is placed on a ring of bearings with a low-pressure air cushion that facilitates rotation of the sphere in any given direction. A smaller sphere is held, by spring supports, against the larger sphere, which allows for a measurement of movement by rotation sensors. These measurements are used to update the images, which are projected onto the surface of the sphere. Thus allowing the user inside to move in any direction.

The creators claim that the CyberSphere combats the problem of movement that is often associated with virtual environments.

The main problem with this idea is if someone is highly claustrophobic, being within a large sphere is not ideal. The idea is sound, as it allows for immersion of the user into the virtual world. Interactability however, could be a problem. The virtual environment surrounds the user, but with the exception of movement, the user cannot interact with it.

The CyberSphere technology allows for movement within the virtual world. This may be applied to our technology in a slightly different manner. To create a sense of movement within the technology, the images may be moved around, it is possible for the user to move the image closer to or further from them as they see fit.

The MRE

In their paper, “Toward the Holodeck: Integrating Graphics, Sound, Character and Story”, the MRE (Mission Rehearsal Exercise) project group outlines their approach to simulating a holodeck.

Having been inspired by Star Trek’s Holodeck, the MRE was formed with the goal of creating virtual reality training environments in which different scenarios may be used [Swa et al., 01]. The main use of the MRE is for military training exercises.

The simulation is hosted in a theatre or cinema type setting. The virtual characters and objects are created on computers and then projected onto a curved screen using three projectors. A sound controller triggers any sound events that are meant to coincide with what is happening in the simulation.

The trainees stand in front of the screen, which provides them with a 150^o field of view. “People participating in the training are immersed in the sights and sounds of the setting and interact with virtual humans acting as characters in the scenario” [Swa et al., 01].

Once again the issue here is the lack of full immersion into the virtual world. Although this project has been proved to be highly successful in training exercises it cannot in the strictest sense be considered a holodeck.

Out of the technologies mentioned thus far, this is perhaps the most relevant to the system proposed in chapter 1. The MRE system makes use of computer graphics quite extensively to create the virtual characters within their simulations. The simulation for our system is completely computer-based, so it might be worthwhile to explore some of the image creation techniques employed by the MRE group.

2.1.3.2 Tele-Immersion

According to Kevin Bonsor, “Tele-immersion is the scientific community' sanswer to the holodeck” [Bon, 02]. Tele-immersion is a combination of cameras and internet telephony, which allows videoconferencing to be taken to the next level. It creates a simulated environment that allows user to come together within a 'virtual room', even if these users are scattered around the world. [Bon, 02] As with most systems of this nature, bandwidth and latency are issues that must be dealt with.

An Initial Testbed for Real-Time 3D-Teleimmersion

A group at the University of Pennsylvania has created a tele-immersion system using off the shelf hardware. The testbed they have created consists of two "tele-cubicles" [Dan et al., 99] hooked via two internet nodes. Each tele-cubicle contains it's own stereo-rig which allows for the realisation of 3D spatial sound. The two dynamic worlds are transmitted over a network, and may be may be viewed with a spatially immersive display consisting of a projector, a tracker and optional 3D glasses. [Dan et al., 99]

The issues of latency and limited bandwidth are dealt with by having a tunable 3D representation whereby the user may decide on which trade-offs are to be made. The goal of tele-immersion applications is the ability to communicate with people who might be geographically distributed, "but are meeting in the space of each local user augmented by the real live avatar of the remote partner" [Dan et al., 99].

Tele-immersion, like our proposed system, does not allow for full user immersion. This is due largely to the fact that the 'worlds' are displayed by projectors onto projection screens, so the user will not be able to get up and actually physically interact with the world. The system proposed by the University of Pennsylvania's team, makes use of avatars, which are graphical representations of the users. These avatars have to be dynamic to allow for realistic motion, which means that the system needs to be able to render them in real time.

2.2 Holography

2.2.1 History

Gabor created his first hologram using a light beam. His holograms were legible, but contained many imperfections because of the light source he used. Laser beams are considered ideal for creating holograms, but were only invented a decade or so after Gabor performed his experiments. [Out et al., 99]

Around 1959-1960 Leith & Upatnieks (at the University of Michigan) reproduced Gabor's holography experiment, but used the laser beam instead of ordinary lamplight. The result was the rebirth of holography into what is now considered modern holography [Out et al., 99].

Holography may be defined as "The technique of capturing, on photo-sensitive material, the image of an object which contains the amplitude, wavelength and phase of the light reflected by that object. The result is a three dimensional image of that object" [Out et al., 99].

2.2.2 Concept

The reason we are able to see objects is because they both emit and reflect the light that arrives at our eyes. A three-dimensional (3D) object scatters light in many directions due to its' inherent three-dimensionality.

To create holograms, we need to reproduce the exact pattern of divergent light, which is created when light is scattered by the actual object. Our eyes would then not be able to distinguish differences between the actual object and the divergent-light pattern sans object [Kra, 95]. This is thus the goal of creating and using a hologram, to make it seem as real as possible.

2.2.3 Photography

The most common method to store and then later recreate an image is to take a photograph. The way it works is that the light-sensitive film within the camera is exposed to incoming light (that has been reflected and emitted off the object), through the camera lens. The negative is thus a two-dimensional (2D) projection of the light field.

A hologram is not a recording of an image as in photography, but rather, the recording of the interference of laser light waves, which are bouncing off the object [Out et al., 99]. Thus enters the need for 3D image capture, and with it the need for holography

2.2.4 Light

Light is inherently a wave, which is a valuable property when one is trying to create a hologram. One is able to specify all the information associated with a wave at a given point by giving its' intensity and phase. So, to record the light scattered by a 3D object, one has to find a way to record both these properties of the reflected light.

2.2.5 Creating Holograms

Conventional Method

The conventional method of creating a hologram requires that a laser beam be split into two. One of the beams after having been spread through a lens falls onto a holographic plate (known as the reference beam), while the other beam is made to spread through the lens thus illuminating the object (known as the object beam). The waves that are reflected from the object end up on the holographic plate where the reference and object beams are interfering. Photosensitive material is then used to record the resulting interference pattern. [Kuj et al., 99]

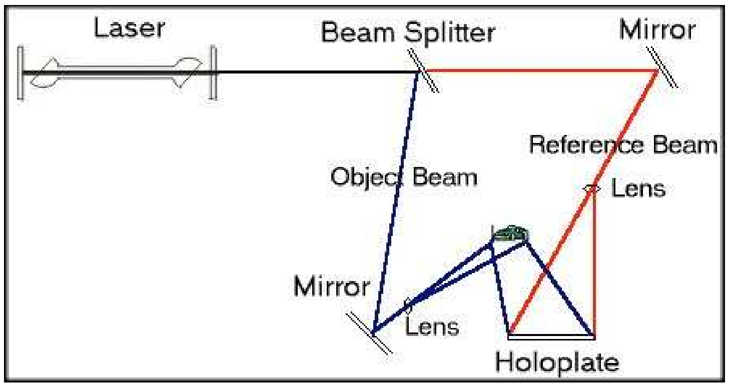


Figure 1. Adapted from [Kuj et al., 99]. This figure illustrates how to record a hologram

Advances in Computer Technology

Due to the ever-increasing sophistication of technology, it is no longer necessary to have an actual object on which to base a hologram. Advances in modern day computing have resulted in the ability to perform ‘ray-tracing’ on computers. Ray-tracing may be used to calculate the pattern of scattered light from any object, hypothetical or not, that one might want drawn and illuminated from any given angle [Kra, 95].

One of the main advantages of using a computer to create holograms is that the computer is able to compute the configuration of the fringe pattern created by combining the light from a direct beam with the scattered light of an object [Kra, 95].

“The holographic aspect of the holodeck is not that far-fetched” [Kra, 95]. The ability to use computers to create holograms out of objects that never existed is a step towards making the holography aspect of the holodeck viable in the near future. The need to have an original image will steadily diminish. This is an interesting fact to note.

The technology we develop is a simulation of a 3D holographic technology. With advances in holographic image projection, images that are created within our virtual holodeck could be projected into space. This cuts out the need for the traditional holographic approach.

2.2.6 Retrieving a 3D image

Once the intensity and phase of the light have been captured, it is possible to recreate an image of the original object by illuminating the film with a source of light that has the same wavelength as the original light. This would allow for the creation of an image of the object at exactly the same position the object was in relation to the film. One then just has to look through the film. Should one move their head to the side, the illusion of looking ‘around’ the edges of the object would be created.

To recreate the hologram we need to make use of a ‘reconstruction beam’, which is made to fall onto the recording medium. The resulting image is that of the real object. Lightwaves are being bent by the interference pattern, causing them to change direction (also known as diffraction). Thus, holograms work as diffraction patterns, which causes the viewer’s eyes to see the diffracted light as though they were looking at the actual object. [Kuj et al., 99]

A computer-generated interference pattern needs only to be projected onto a screen that is illuminated from behind. The result is a 3D image of an object that quite possible never existed. [Kra, 95]

2.2.7 State of the Art

The TransScreen

The TransScreen (TS) is in effect a very large screen that allows for images to be projected onto it. The screen is composed of a microscopic pattern of particles that are suspended in a dense medium. These particles allow for the simultaneous diffraction, reflection and transmission of all wavelengths of light. This is to accommodate viewing of both the image and through the screen. Polarised or non-polarised projectors may be used to project images onto the screen. The particles within the TS allow for “Polarisation preservation”, which means that stereoscopic projection is possible with the use of two projectors, one for each eye. [Las, 02]

The TS makes it possible to create the illusion of 3D depth by making use of “depth cues” [Las, 02]. An important element in 3D perception the ability of the eye to focus on both near and far objects. The TS takes advantage of this by allowing the eye to see the projected video image as well as a person or object behind the screen simultaneously. By making use of this ability, the eyes tell the brain that they are able to see in 3D. [Las, 02]

The TS is able to accommodate both front or rear projected film or video images. The images are partially captured by the TS’s imaging medium, making the projected image clearly visible. The TS also allows the scene from the environment behind the screen to be clearly visible in the area of the screen in which no image is projected. The image that is brighter (projected or environmental) is clearly visible to the observer. [Las, 02]

The TS has recently been used in Stephen Spielberg's film "Minority Report". The TS is a most novel idea, and one that clearly works. The problem with the TS is that to display images a special screen, the TS, is required. The aim of the holodeck is to obtain images that can be projected into the middle of a room without the need for any object onto which the image needs to be projected. To have pieces of screen scattered around a room can be quite hazardous, they will also interfere with the ability to interact with the environment. The ability to project free-floating images is a break-through in the field of holography though.

Actuality's Globe-Shaped Display

The display unit is a glass dome, the same as that those of streetlight covers. A paper-thin transparent screen, that can rotate at 730-revolutions per minute, is contained within the dome. An image coming in from a workstation is manipulated and broken upto into 'slices'. These slices, when projected onto the rotating screen, create a spherical image which appears to be free-floating within the dome. [Ber, 02]

The current prototype of the display projects an image up to 10 inches in diameter, and is made up of 100 million voxels. The display could potentially be adapted to display images as large as 2 feet in diameter. [Ber, 02]

The HoloTank

The HoloTank is a 3D volumetric display, similar to an aquarium, filled with liqud that contains proprietary microscopic particles. Images are displayed by projecting scanned laser light or vector video graphics into the display. Animated vector graphics, such as drawings, logos and cartoons may be used to create 3D holographic type projections. [Las, 02]

The problem with an aquarium-type approach is that in order to create a 'world' in which a user could interact, one would have to give the user scuba gear. This is however, impractical as this would prove to be very cumbersome, and would restrict any interaction with the virtual environment, besides according to Halle "air, water, or smoke are, in general, very poor display media. "[Hal, 97]

NASA's Holographic Display Project

The main purpose of this project was to illustrate that 3D virtual reality displays can be viewed without using visual aids. Older display technologies require the use of stereoscopy to give the appearance of three-dimensionality. This new technique invented by NASA, does not involve the use of polarising goggles, goggles equipped with mini video cameras or any other visual aids [Nas, 02].

The visibility if the image will not be restricted to a narrow range of directions about a specific line of sight to a holographic project plate. Rather, the image is visible from any side, or from the top, ie the image is visible from any position with a clear line of sight to the projection apparatus (the image may be viewed as though it were really a 3D object) [Nas, 02].

The special 'ingredient' of this approach, is the use of a block of silicar aerogel as the display medium. The aerogel is a form of open-cell glass foam that has a similar chemical composition to quartz, but it is approximately one-tenth the density of quartz. The aerogel is a suitable display medium because it is almost completely transparent, but allows for both the reflection and the scattering of light to produce a real image. [Nas, 02]

There are three ways in which a hologram could be displayed: by using 2D images, by using computer-generated images, or by using a combination of both. "One could use static holograms to project still images, either alone or in combination with computer-generated holograms to project moving or still images. A computer-generated hologram would be downloaded into a large liquid-crystal, which would be illuminated by a laser projection apparatus to display the holographic image in the aerogel block"[Nas, 02].

The advantage of having aerogel as your projection medium is that you can allow for both static projection and animated projection at the same time. The disadvantage is once again the lack of interaction afforded by this approach. It is not easy to interact with a block, there is no way to become immersed in the environment.

Autostereoscopic Technology

In order to create a proper holodeck, advances in digital storytelling, gaming, artificial intelligence, human interface design and other technologies need to be accomplished [Jac, 01]. Jacobs notes that research institutes such as NYU and MIT have been experimenting with autostereoscopic technologies for the last decade or more. The basic requirement for autostereoscopic displays is that the screen should be able to show each eye a different image, this is similar in concept to 3D postcards.

Autostereoscopic displays present viewers with 3D images without the need for any viewing aids. Halle also states that 3D displays have recently become both popular and practical in the computer graphics community. [Hal, 97]

Mark II Holographic Video

A spatial imaging group, headed by Stephen Benton at MIT is currently working to perfect their "Mark II Holographic Video" [Fre, 02] system. The system makes use of both an adaption of the classic method of creating holograms (see figure 1), and a series of specially developed computer algorithms desgined by Benton and his team. The algorithms are used to calculate the kinds of microscopic lines required to create the hologram. The algorithms convert these lines into sound waves and passes these waves through a stack of tellurium-oxide crystals. The crystals are renowned for having the ability to temporarily distort when sound waves are passed through them. The distortion allows for the formation of the diffraction pattern required to make up the hologram. The image from the crystals is projected onto a view screen by passing a laser beam through the diffraction pattern caused by the aforementioned distortion. [Fre, 02]

Several of the technologies mentioned above are not directly related to our project, they are however, relevant to our area of interest. Many of them pose interesting options for doing certain things, eg considering that the holodeck is partially a volumetric display medium, Nasa's project using aerogel as a volumetric medium as well as the Holotank which is also a volumetric display, pose alternatives to be considered.

2.2 Rendering Algorithms and Hardware

To begin understanding how 3D graphics work it is necessary to understand how rendering algorithms and hardware work. The graphics pipeline illustrates what an object has to go through before being displayed on the computer monitor.

2.2.1 The Graphics Pipeline

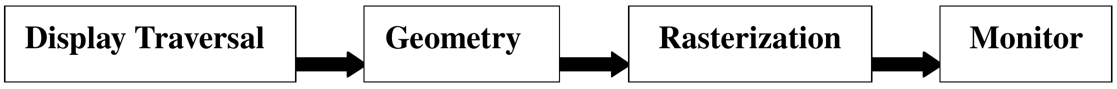


Figure 2. Adapted from [Fol et al., 01]. A Graphics Pipeline

Each of the blocks above represents a stage in the graphics pipeline. Each stage will be explained in a fair amount of detail in the following sections

2.2.1.1 Display Traversal

The first thing that needs to be done is a traversal of the display model, or database. This is necessary to ensure that the latest image is retrieved (images may change from one from frame to another). All the primitives contained in the database must be fed into the rest of the pipeline, along with context information (such as colour and transformation matrices). [Fol et al., 01]

There are two main forms of traversal, retained mode and immediate mode. A display processor can handle retained mode if the structure database is stored in the display processors' local memory. The immediate mode traversal must be performed by the cpu, which causes the cpu to use up cycles which could be otherwise used. [Fol et al., 01]

2.2.1.2 Geometry

The geometry phase of the graphics pipeline is concerned primarily with transformations and image culling.

- Transforms (also known as Affine Transformations [Hil, 90]) - These are the four primary transforms: Translation, Rotation, Scaling and Skewing (Shearing).

  1. Translation is defined as the straight line movement of an object from one position to another [Hea et al. 86]. The movement of the object may be along any of the three axes. The mathematics involved would normally be simple addition and subtraction, but has been adapted for efficiency purposes, to involve matrix multiplication. [Sal, 01]

  2. Rotation may defined as the transformation of object points along a circular path [Hea et al. 86].

  3. Scaling is defined as a tranformation to alter the size of an object [Hea et al. 86]. Scaling may be used in creating the effect of depth of field during perspective projection [Sal, 01].

  4. Skewing (Shearing) is defined as the transformation to use when changing the shape of an object. The transformation occurs by manipulating the object along one of the axes. [Sal, 01]

- Trivial Accept/Reject Culling [Sal, 01] - The main goal of this step is to reduce the number of triangles that need to be rendered. To decide whether or not a triangle is within the view frustum, a simple test is performed to check if the x, y and z coordinates of each of the triangles vertices are within the frustum. If not, the triangle does not get rendered. If, however, a piece of the triangle is visible, the polygon formed (the pieces out of the frustum are effectively cut off) is re-tesselated (subdivision into triangles) is performed. The test is then performed again.

- Lighting - This operation usually takes place once the 3D scene has been placed into view space. Geometric lighting is applied to objects in the scene. Geometric light is based on simple lighting and reflection models which, on the whole, have very little in common with real world lighting, but which are deemed adequate for 3D, real-time rendering. [Sal, 01]

  There are two types of illumination models, local and global. The local model considers objects in isolation, it is a simple model which allows for quick rendering but does not allow for shadows or reflection. The global model considers the effect of objects on other objects with a scene. It also allows for the incorporation of shadows, reflection and refraction. [Ban, 02]

  The two types of lighting may be further broken down into three types of light [Sal, 01]:

  - Directional lights are global, and are considered to be infinitely far away. A real world example of this would be the sun.
  - Point lights are local lights which are defined points in space which emit light in all directions.
  - Spot lights are also local sources of light which are defined points that emit light in a specified direction in a cone-shape.

- Clipping – This step involves checking for lit primitives that are not trivially accepted or rejected, and then clipping them to the view plane. This step serves two main purposes: to prevent activity in one screen window from affecting pixels in other windows, and prevention of mathematical overflow or undeflow from primitives that pass behind the eye or very far away. [Fol et al., 01]

2.2.1.3 Rasterization

- Shading – There are three main types of shading – Flat (or Constant), Gouraud and Phong.

- **Flat (or Constant)** – Generally this is not very realistic shading, the exception is when the object has no surface designs, textures or shadows [Hea et al. 86]. The renderer takes the colour values from each of the triangles vertices and averages them out to get a shading colour which is applied to the whole triangle [Sal, 01].

- **Gouraud** – Named after it's inventor Henri Gouraud. Makes use of bilinear intensity interpolation. [Wat, 90] It works by removing intensity discontinuities between adjacent planes of a surface representation by linearly varying the intensity of each of the planes so that the overall result matches the intensity values at the plane boundaries. [Hea et al. 86] Gouraud shading removes intensity discontinuities (associated with flat shading), but highlights on the surface sometimes appear as deformed shapes, and linear intensities can cause bright or dark streaks (known as Mach Banding). [Hea et al. 86]

- **Phong** – Named after it's inventor Bui-Tuong Phong, Phong interpolation overcomes some of the disadvantages experienced with Gouraud shading. Bilinear intrpolation is also used, but the attributes that are interpolated are the vertex normals rather than the vertex intensities. Separate intensities are evaluated for each pixel from the interpolated normals. [Wat, 90]

- Antialiasing – A common problem with undersampled scenes is aliasing or "jaggies" [Hil, 90]. To correct this problem, a form of blurring is applied to smooth the image out (supersampling, prefiltering or postfiltering). One way of rectifying this is to add varying shades of the contrasting colours, so when the image is seen far away, the eye automatically blends them resulting in a smoother looking image [Hil, 90].

- Ray Tracing - ray tracing is the process of creating 3D images through the use of complex light interactions. Ray tracing attempts to simulate the path rays of light will take when bouncing around the 3D world. The goal of ray tracing is to determine the colour of each light ray that hits view space before reaching the eye. [Rad, 01]

- Rendering Primitives – The actual rendering of primitives occurs in this stage as well. Some of the more well known line drawing methods are outlined in the following sections:

- **Conventional Line Algorithms**

  The most straightforward approach to drawing a line would be to use the mathematical line formulae.

$$y = mx + c \quad (1)$$

  The problem with computer displays is the fact that all images are created out of a set of 'highlighted' pixels. The pixels only have integer coordinates, which means that using the standard line formula the computer has to perform rounding operations on each set of coordinates it calculates. Rounding operations performed result in computational overheads. There are two main solutions to this problem, the Digital Differential Analyser (DDA) and Bresenham's line algorithm. [Hea et al., 86]

- **The Differential Digital Analyser (DDA)**

  The DDA calculates pixel positions along a line by using:

$$\Delta y = m*\Delta x \quad (2)$$

  The DDA takes unit steps with one coordinate and calculates the corresponding values for the other coordinate. [Hea et al., 86]

If the slope of the line is less than or equal to one, the DDA takes the change in the x-coordinate to be one. If, however, the line's slope is greater than one, the DDA takes the change in the y-coordinate to be one. Should the slope be negative we apply either of the above methods (depending on whether the slope is greater than negative one or lesss than or equal to negative one), but make the change negative one. [Hea et al., 86]

The DDA algorithm is faster than using the ordinary line formula, but suffers from the division operations performed to find the incremental value used to calculate the next pixel. [Hea et al., 86]

- **Bresenham's Line Algorithm**

  Bresenham's line algorithm makes use of only integer arithmetic, which means that it improves on both the conventional formula and the DDA. There are no real (double or float) variables used, which means that there is no need for rounding. Bresenham's algorithm makes use of a decision variable to decide which pixel is chosen to be the next one. [Fol et al., 84]

  Bresenham's algorithm is considered an important example of an "incremental algorithm" [Hil, 90] that makes use of information about the previous pixel to calculate the position of the current pixel. [Hil, 90]

2.2.1.4 Monitor

This stage handles image display. This phase has two main functions – Gamma correction and Analogue to Digital Conversion (ADC).

- Gamma Correction – This is performed to ensure the image is corrected for any nonlinear relationships between pixel values and displayed intensities. [Lev, 02]

- ADC – This is performed so that we see the images. To project the image onto the screen, it has to be converted from an analogue signal to a digital one.

2.2.2 Graphics Cards

This section serves as a brief introduction into how graphics are made up, as well as how they display images. The basic components that make up a graphics card are memory, a computer interface and a video interface.

- Memory - This memory is for use by the frame buffer to store the complete bit-mapped image that will be sent to the monitor. [Web, 02]
- Computer Interface – This allows the cpu to modify the contents of the frame buffer. The link is usually established by plugging the graphics card into the motherboard. [Din, 02]
- Video Interface – This causes the generation of signals for the monitor and or other output device. The card must be able to generate analog signals. This is done by using the RAMDAC (Random Access Memory Digital to Analog Convertor). [Din, 02]

Moving onto how images are displayed. The CPU sends small instruction sets to the frame buffer, which are interpreted by the graphic card's driver and executed by the graphics processor (gpu). Operations such as bitmap transfers, window resizing and positioning, scaling and polygon drawing can be handled by the gpu. [Din, 02]

Graphic card drivers translate what the application wants to display into instructions the gpu can understand. Newer graphic cards support drawing operations for complex primitives, as well as the movement of large blocks of information. It is then up to the driver to decide when or whether these extra functions will be used, as well as how they will be used. [Din, 02]

Chapter 3

To implement a holodeck, it is necessary to devise a holographic technology. This technology takes the form of creating images solely out of spheres. Considering the 3D nature of the holodeck, the main options for a basic component are cubes or spheres. Cubes are in some respects similar to the pixel approach, whereas spheres have not been used as building ‘blocks’ before.

Most 3D objects are represented through the use of primitives, when the primitives are rendered an image of the object is the result. To accommodate this, the holodeck is able to render several primitives (lines, triangles, tetrahedrons, polygons and pyramids). Each of the primitives is made of spheres.

The sphere is the smallest possible unit available in the holodeck, as opposed to the more conventional pixel approach. As mentioned in the Introduction, we assume the ability to render spheres anywhere within the holodeck.

3.1 System Architecture

3.1.1 Image Creation

An object that is to be displayed within the holodeck has to follow certain steps. These steps involve reading an object data file, converting the object primitives into 'holodeck' primitives, recording the sphere information, and then rendering the image within the holodeck environment. Figure 9 shows a 'pipeline' of what the steps are, and in which order they come.

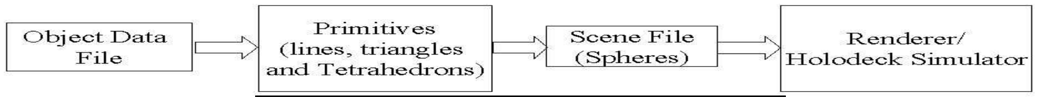

Object Data File
Primitives (lines, triangles and Tetrahedrons)
Scene File (Spheres)
Renderer/ Holodeck Simulator

Figure 3. The image creation and display pipeline

To some degree Figure 3 is similar to the graphics pipeline (section 2.2.1). The graphics pipeline stages are Display Traversal, Geometry, Rasterization and Monitor.

The Stages of Image Creation

- Object Data File – This can be likened to the Display Traversal stage of the graphics pipeline. Both stages perform object information retrieval from a source, whether a file, database or dislay model. In this case, retrieval of object information is from a file.
- Primitives – This stage is equivalent to the Geometry and Rasterization stages combined. The reason for this is that the transformations, lighting, depth testing and shading occur within this stage.
- Scene File – This is stage performs the intermediate storage of image information before it is rendered.
- Renderer/Holodeck Simulator – This stage is similar to the Monitor stage of the graphics pipeline. A major difference between the two, however, is that the Monitor stage is exactly that, image display on a monitor. Even though we are restricted by 2D displays, this stage represents the image display within a volumetric environment.

Controlling Image Creation

To control the image creation process, use is made of the following three programs: CreateSceneFile, DrawSpheres and OFFConverter. Each of the programs handles a separate section of the pipeline in Figure 3. To gain a better understanding of each program and the purpose it serves, each will be discussed individually.

- OFFConverter – The main aim of this program is to retrieve 3D object information from a file and rewrite the information in a format in which the next program can understand. Another important function this program performs, is that of normal calculation. For each of the primitives read, this program will calculate the normals of the face of the primitive. These normals also get written to the file that is passed to CreateSceneFile.
- CreateSceneFile – This program receives input from the OFFConverter via files. Each file is read, extracting the coordinates and normals of each primitive into temporary variables for calculations. The calculations include working out how many spheres are required to create each primitive, as well as the shading information required for each sphere. The shading information is incorporated into the colour values of the sphere (refer to section 3.3). The sphere information (x, y and z coordinates, radius, red, green and blue colour values) is written to another file type to be read by DrawSpheres.
- DrawSpheres – This program reads the file passed to it, extracting and then placing within the holodeck the spheres to create the object primitives. This program also provides extra lighting effects within the holodeck.

3.1.2 Spheres

To give more control over image creation, it is deemed necessary that each sphere has several properties:

- A set of coordinates (x, y, z) – These are for the placement of the sphere within the holodeck.
- Radius (size) – The size of the sphere is a necessary attribute for two reasons, firstly it affects the number of spheres in a primitive, and secondly it affects image quality.
- Colour (r, g, b) – These values are perhaps not as important as the others, the reason being that the system could just set it's own colours and make use of those values. The reason they

are in the system is to allow for the easy change of colour. Another purpose they serve is that of containing the values of the shading that is performed on each sphere.

- A quality value – The quality value is more a global value, although it does impact the spheres directly (refer to section 3.1.3).

For future enhancements:

These values would be nice to allow for more complex lighting and shading.

- *A transparency value*
- *An emission value, and*
- *A reflectivity value*

3.1.3 Quality

To allow the user to control the quality of the image produced, we introduce the concept of the 'quality value'. One reason for the quality value is the trade off between quality and time to render, refer to chapter 5. Basically, this allows the user to decide where the trade off happens. The lower the quality of the image, the faster the image renders.

The quality value of the spheres serves as a scale:

- Between 0 and 1: The spheres do not touch, which gives the appearance of gaps in the image.
- Exactly 1: The spheres just touch, which results in lines that appear quite bumpy because of the spheres.

Greater than 1: The spheres overlap, which results in lines that get closer in approximation to straight lines, ie there are less bumps.

3.2 Primitives

3.2.1 Lines

To construct lines out of spheres, it is necessary to adapt one of the above algorithms. Considering the nature of the task, Bresenham's line algorithm is not a good choice, as we will not be able to use only integer arithmetic, the reason being that we will be adding a ratio (refer to the algorithm below) which is a floating point number. If the algorithm was to use integer arithmetic, the values to be needed to get the next sphere coordinates would be rounded. This would result in rounding errors, and inaccurate lines, as the values required are seldom whole numbers.

Instead, an algorithm such as the DDA may be reworked for our purposes. The main difference is that we will no longer be calculating which pixel to colour in, but rather where the next sphere will go. In order to this, it is necessary to calculate the distance between the two points and divide it by the radius size of the sphere. To incorporate the quality factor mentioned earlier, it is necessary to multiply the number of spheres by the quality value. The result is a ratio that is used to calculate each of the coordinates (x, y, and z) for each sphere. The coordinates calculated become the center point of the sphere. The result is a line drawing algorithm that draws 3D lines, whereas Bresenham's algorithms and the DDA are 2D line drawing algorithms.

```
function drawLine (Start_Point, End_Point)
    //Change in the x, y and z coordinates
    dx = End_Point.x – Start_Point.x
    dy = End_Point.y – Start_Point.y
    dz = End_Point.z – Start_Point.z

    distance = Square Root($dx^2 + dy^2 + dz^2$)  //Pythagoras theorem
    Number_ of_ Spheres = quality*distance/radius

For 1 going to Number_of_Spheres
    Record sphere information
    x = x + dx/Number_of_Spheres
    y = y + dy/Number_of_Spheres
    z = z + dz/Number_of_Spheres
End For
End Function
```

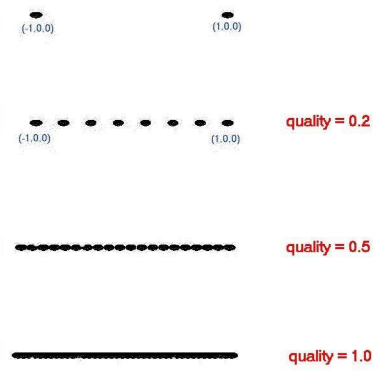


Figure 4. The same line at different levels of quality

3.2.2 Triangles

According to Salvator [Sal, 01], triangles are the most common rendering primitives in standard rendering algorithms. It is therefore necessary to have a triangle primitive that is made out of spheres for use within the holodeck. The triangle primitive is often made use of when using 3D objects. Our system makes use of two types of triangle, a frame triangle and a filled triangle.

3.2.2.1 Frame Triangles

Frame triangles are essentially three lines joined together to form a triangle. They are best to use a wireframe of an object is required.

3.2.2.2 Filled Triangles

We examine several approaches to creating filled triangles. The first approach draws a frame triangle, and then draws lines from the apex to each of the sphere coordinates on the base line.

```
Function FilledTriangle1 (Point1, Point2, Point3)
        Decide which point will serve as the apex
        Calculate how many spheres are required to draw the line opposite the apex
        Create an array to hold the coordinates of the spheres that will make up the line opposite
        the apex
```

```
    Calculate the coordinates for each sphere and store these coordinates in the array

    While there are still spheres to draw
        Draw a line from the apex to a point contained within the array of spheres
End While
End Function
```

The problem with this approach is the resulting image. At high levels of quality this approach works fairly well, but as the quality is reduced it becomes obvious that this approach is unsuitable. The reason being that the area around the apex has a heavy concentration of spheres, while the effect tapers out towards the bottom of the triangle.

The second approach focuses rather on calculating the areas of both the triangle and the sphere and recursively dividing under the first until it is comparable with the second.

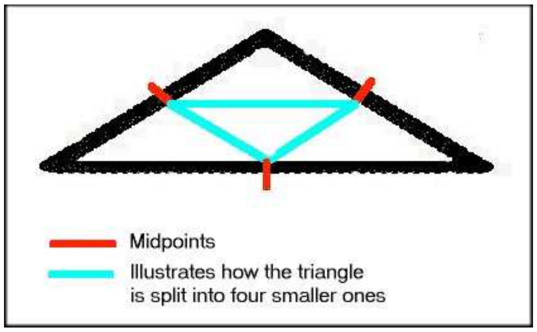
```
Function FilledTriangle2  (Point1, Point2, Point3)
Calculate the area of the triangle
Calculate the area of the spheres to be used (Area = $\Pi r^2$)

If the triangle area is greater than the sphere area
    Calculate the midpoint of each line of the triangle, using the following approach:
     newPoint1.x = (Point1.x+Point2.x)/2
     newPoint1.y = (Point1.y+Point2.y)/2
     newPoint1.z = (Point1.z+Point2.z)/2

    //Break the triangle into four smaller ones  (See Figure 5)
    FilledTriangle2 (Point1, newPoint1, newPoint2)
    FilledTriangle2 (newPoint1, newPoint2, newPoint3)
    FilledTriangle2 (Point2, newPoint1, newPoint3)
    FilledTriangle2 (Point3, newPoint2, newPoint3)
```

```
Else
        Record sphere information
End If
End Function
```

This algorithm, however, is still lacking. The reason being that it does not allow for variation in quality levels. To overcome this problem, we change the way the algorithm works.



Midpoints
Illustrates how the triangle is split into four smaller ones

Figure 5. Diagram illustrating how a triangle is broken up into four smaller one

To fill the triangle, the algorithm (after calculating the triangle's area and the sphere's area) first calculates how many recursions are needed using equation 3.

$$\text{num\_Recursions} = \text{quality} * \log_4(\text{triangleArea}/\text{sphereArea}) \quad (3)$$

Equation 3 requires further explanation. To understand how a triangle is broken up into four smaller ones by the system, refer to Figure 5. The reason then that the above equation is $\log_4$ is because the triangle is broken up into four smaller triangles. The ratio (triangleArea/sphereArea) gives how many spheres will fill the triangle. The quality factor is also used in the calculation to give control over the fill of the triangle. The higher the quality value, the more spheres are used to fill up the triangle.

```
Function FilledTriangle3  (Point1, Point2, Point3, Normal_x, Normal_y, Normal_z,
                           number_of_Recursions)

Calculate the area of the triangle
Calculate the area of the spheres to be used (Area = $\Pi r^2$)
    Calculate number_of_Recursions using equation 3

While number_of_Recursions is not zero
    Calculate the midpoint of each line of the triangle, using the following approach:
     newPoint1.x = (Point1.x+Point2.x)/2
     newPoint1.y = (Point1.y+Point2.y)/2
     newPoint1.z = (Point1.z+Point2.z)/2

    //Decrement number_of_Recursions
    number_of_Recursions = number_of_Recursions - 1

    //Break the triangle into four smaller ones  (See Figure 5)
    FilledTriangle3 (Point1, newPoint1, newPoint2, Normal_x, Normal_y, Normal_z,
                     number_of_Recursions)
    FilledTriangle3 (newPoint1, newPoint2, newPoint3, Normal_x, Normal_y,
                     Normal_z,  number_of_Recursions)
    FilledTriangle3 (Point2, newPoint1, newPoint3, Normal_x, Normal_y, Normal_z,
                     number_of_Recursions)
    FilledTriangle3 (Point3, newPoint2, newPoint3, Normal_x, Normal_y, Normal_z,
                     number_of_Recursions)

End While

Calculate sphere coordinates by taking the average of the three corners of the triangle, so that
the sphere ends up in the center of the triangle
```

```
//Calculate the weights required to shade the sphere (section 3.3), the weights are stored //in a global array
GetPointWeights (Point1, Point2, Point3, sphereCenter.x, sphereCenter.y,
                 sphereCenter.z)
DrawPointPhong (Point1, Point2, Point3, Normal_x, Normal_y, Normal_z,
                sphereCenter.x, sphereCenter.y,  sphereCenter.z)
Record the sphere information

End Function
```

The third approach to creating filled triangles works the best, the result may be seen in Figure 6.. It manages to fill the triangle and incorporate quality into its workings. A problem we have encountered with it though, is that to get triangles that are filled with a large quantity of spheres, the quality level has to be set to one or higher. The reason for this may be seen if we examine equation 3 again. The quality value is multiplied by the result of the logarithmic expression, which depends on the areas of both the triangle and the sphere.

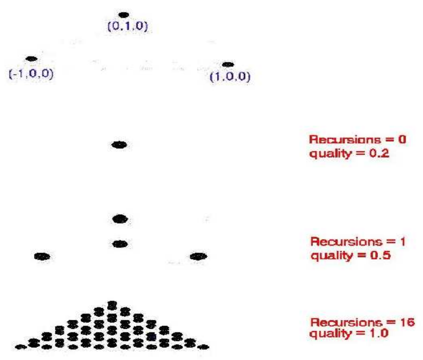


Figure 6. The same triangle at various levels of quality

3.2.3 Tetrahedrons

As with the triangles, our system makes use of frame tetrahedrons and hollow tetrahedrons (where each face is a filled triangle). Another type of tetrahedron available is the 'solid' tetrahedron.

3.2.3.1 Frame Tetrahedrons

The frame tetrahedron is simply four frame triangles connected at the vertices.

3.2.3.2 Hollow Tetrahedrons

The hollow tetrahedron is four filled triangles connected at the vertices. The need for a hollow tetrahedron arises out of the requirement to have solid-looking images within the holodeck. But, being hollow still allows for relatively fast rendering times, as the tetrahedron just appears solid.

3.2.3.3 Solid Tetrahedrons

Considering that a tetrahedron is a 3D primitive, it is difficult to find an effective way of creating a solid tetrahedron. The reason for this is that it becomes necessary to work with the volumes of both the tetrahedron and sphere that is needed to fill it.

To get a working algorithm for a solid tetrahedron, once again, involves trying several approaches. The first approach is loosely based on the first attempt of the filled triangle. The similarity being that an attempt is made to split the tetrahedron up into a set of filled triangles.

Firstly, this does not give the proper result needed, as the quality would always have to be at maximum to get a realistically filled tetrahedron. The reason for a constant maximum quality arises out of the deficiency of *FilledTriangle1* (refer to section 3.2.2.2). If anything less than one is used, there will be a large concentration of spheres at the apex of the tetrahedron, while near the base, the density of spheres will begin to thin out. Having maximum quality then defeats the objective of having a quality value that may be changed.

Secondly, having to do this results in very long rendering times as there are too many spheres to render.

To solve these problems, another approach is adopted. This approach fills the volume of the tetrahedron with spheres. This involves breaking the tetrahedron into smaller ones that may be dealt with individually. If each of the corners is ‘removed’, the result is a diamond shape that makes up most of the center of the tetrahedron. There are several ways to fill the space occupied by the diamond; two of these approaches are explored here.

Approach one involves filling the center of this diamond with one large sphere, and then calculating how much space is left to fill using smaller spheres. Using this approach causes substantial computational overhead because the function would have to calculate the distance from the center of the large sphere to each of the vertices. The function then needs to subtract the radius from each of these distances, which shows how much space is still available. Thereafter, the function needs to calculate the midpoint of the remaining distance, to allow for the placing of more spheres, while ensuring they do not go out of the boundaries.

Approach two is decidedly a lot simpler to implement. To fill the diamond with spheres, break the diamond down into two pyramids, which are broken down into tetrahedrons, so this is effectively more recursion. This approach has a higher complexity than one, but is a lot easier to implement.

Approach two is the one that is used within the method:

```
Function SolidTetrahedron (Point1, Point2, Point3, Point4)
    //Calculate the area of the tetrahedron
    Get vector1 between Point1 and Point2
    Get vector2 between Point1 and Point3
    Get vector3 between Point1 and Point4
    Find the crossproduct of vectors 2 and 3
```

```
        Find the absolute value of the dotproduct of vector1 and the crossproduct and multiply by
        1/6

        //Find the radius of to use for the sphere
        newRadius = Cube Root ((tetrahedronVolume / 2.3) /Π);

If the new radius size is greater than the old radius size * quality then
        Calculate the midpoint of each line, storing the points in temporary variables (p1Top2,
        p1Top4, p1Top3, p2Top3, p3Top4, p2Top4)
        Take the average of the x, y and z values of the new points which are stored in the
        temporary variables to get the sphere coordinates

        solidPyramid (p3Top4, p1Top4, p2Top4, p2Top3, p1Top3)
        solidPyramid (p1Top2, p1Top4, p2Top4, p2Top3, p1Top3)

        solidTetrahedron (point1, p1Top2, p1Top4, p1Top3)
        solidTetrahedron (point2, p1Top2, p2Top3, p2Top4);
        solidTetrahedron (point3, p2Top3, p3Top4, p1Top3);
        solidTetrahedron (point4, p1Top4, p2Top4, p3Top4);
End If
```

This algorithm is proving to be quite effective at producing solid tetrahedrons. One problem noticed is a slight bulge showing through the base of the tetrahedron. One other problem is the computation cost of this algorithm, it is much more efficient than the first approach, but still has perform several computations to find an optimal radius size.

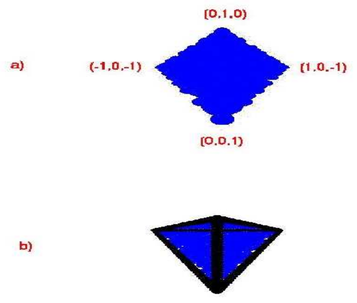


Figure 7. Filled Tetrahedrons. a) With a frame, b) Without a frame

3.2.4 Remaining Primitives

Unlike lines, triangles and tetrahedrons, the remaining primitives (pyramids and polygons) are relatively uninteresting. The pyramid is basically two tetrahedrons connected (distinction may be made between solid, hollow or frame), while the polygons are made up by using either a combination of lines and triangles, or by using four tetrahedrons joined together.

3.3 Shading

To get a degree of realism, it is necessary to explore shading techniques. An adapted version of Phong shading algorithm is used for shading. Phong shading works by interpolating vertex normals across the face of a polygon or triangle. The Phong lighting model usually illuminates the pixel at each point. [Ham, 99]

An assumption of the shading method is that if the spheres are shaded correctly, then the primitive is shaded correctly.

Some of the limitations associated with our shading method are highlights, textures and reflection coefficients. The reason for this is that have not been implemented, it is however feasible to incorporate them at a later stage.

Our system works with spheres as the basic component of any given image, so this needs to be catered for in the shading algorithm. It is no longer necessary to calculate the intensity levels for each pixel, but rather the intensities need to be calculated for each individual sphere.

To calculate the intensity of each sphere, it is necessary to determine the position of the sphere (it's center) as a weighted sum of the vertices; there are three weights for each point (this stands to reason because of the three vertices). This is required so that a smooth shading effect is achieved throughout the triangle. Some assumptions the calculations are based on:

1. $0 < w_i < 1$ - Each weight is between 0 and 1. The only reason a weight would be 0 or 1 is because the sphere is at the position of one of their vertices

2. $w[0] + w[1] + w[2] = 1$ - The three weights added together should give a value of one.

3. $w_i(P) = w_{ix}*x + w_{iy}*y + w_{iz}*z$ - The weights are functions of the positions of the points, so it stands to reason that when calculating $w_{0,}$ $w_0(P_0) = 1$ (and likewise with w_1 and w_2).

Based on assumption 3 above, there are three resulting equations to be solved in order to obtain each weight, for illustration purposes, the whole process will be explained from beginning to end for weight 0 (w_0):

4. $w_{0x}*P_{0x} + w_{0y}*P_{0y} + w_{0z}*P_{0z} = 1$
5. $w_{0x}*P_{1x} + w_{0y}*P_{1y} + w_{0z}*P_{1z} = 0$
6. $w_{0x}*P_{2x} + w_{0y}*P_{2y} + w_{0z}*P_{2z} = 0$

To calculate w_0 it becomes a simple matter of solving the above equations simultaneously.

$$w_{0z} = \frac{P_{1x}*(P_{0y}*P_{2x} - P_{2y}*P_{0x}) - P_{2x}*(P_{0y}*P_{1x} - P_{1y}*P_{0x})}{((P_{0y}*P_{2x} - P_{2y}*P_{0x})*(P_{0z}*P_{1x} - P_{1z}*P_{0x}) - (P_{0z}*P_{2x} - P_{2z}*P_{0x})*(P_{0y}*P_{1x} - P_{1y}*P_{0x}))}$$

$$w_{0y} = \frac{(P_{2x} - w_{0z}*(P_{0x}*P_{1z} - P_{1x}*P_{0z}))}{(P_{0y}*P_{2x} - P_{2y}*P_{0x})}$$

$$w_{0x} = \frac{(1 - w_{0y}*P_{0y} - wz*P_{0z})}{P_{0x}}$$

$$w_0(P) = w_{0x}*x + w_{0y}*y + w_{0z}*z$$

Similarly, it is possible to calculate w_1 and w_2. The difference being that when w_1 is calculated, $w_1(P_1) = 1$, while for w_2, $w_2(P_2) = 1$.

Using the above weight calculations, it is now possible to adapt the shading.

```
// OFFConverter program, this is not the full algorithm, only the relevant piece
Read triangle vertices from OFF file
Calculate the normals for each triangle
Write the triangle coordinates and normals
```

Please note, the following methods are called within the primitive drawing methods in the CreateSceneFile program.

```
//GetPointWeights method
Receive the points of the triangle, as well as the x, y and z value of the current sphere
Calculate the weights for the current sphere using the above equations.

//intensity
Receive basic vectors, normals and material reflectivity values from the calling method
Calculate light intensities using the lighting equation:
```

$I = ambient_{Mat} + diffuse_{Mat} * diffuse + specular_{Mat} * specular^{Matexponent} \quad (4)$

```
 //DrawPointPhong
Receive the points of the triangle, as well as the normals calculated by the OFFConverter
and the coordinates of the current sphere
Using the weights calculated above and the triangle normals, calculate the normals for
the current sphere coordinates (this is done by taking the x (then y and z) value of each point and
multiplying it by it's equivalent weight ie P1.x * [0])
Call the lighting routine which determines values for the red, green and blue components of the
shading.
```

To illustrate the shading concepts described above, refer to Figure 8. For Figure 8, the lighting was setup as follows:

- Ambient is red
- Diffuse is yellow
- Specular highlights are blue

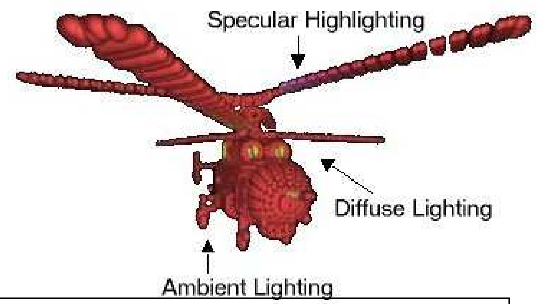


Figure 8. A shaded helicopter

3.4 Controls

A graphical user interface (GUI) is used to allow the user to control the image creation process. The GUI combines the functionality of the ‘OFFConverter’ program and the ‘CreateSceneFile’ program, and makes calls to a standalone version of the ‘DrawSpheres’ program. The main reason for this is that the two former programs handle the conversion process, while the latter program handles only display and image manipulation. Another reason is that it is not necessary to have the GUI running in order to view an image, provided the scene file to be loaded already exists, the user may use the ‘DrawSpheres’ program with the scene file name as a command line parameter.

The system GUI is split into three sections. The first section allows the user to convert objects from OFF files to scene files. The second section allows the user to convert intermediate file objects or test file objects into scene files (refer to section 4.1). The last section allows the user to view the images from the objects that are converted, by calling the DrawSpheres program.

The ‘DrawSpheres’ program has also got a GUI, which is used for changing the image view. Alternatively, the user can change the view using keyboard commands. This GUI is not divided into sections like the main system GUI, the only functions that are in their own section are the controls to enable or disable BMRT (refer to section 4.3).

Chapter 4

In this chapter, we discuss the implementation of image creation programs, as well as the GUI's that are available. We also discuss some of the finer details of the programs such as why certain checks are in place, or why something is implemented the way it is.

The entire project is implemented using a combination of C++ for the calculations, and openGL, which is the current industry standard (an exception will be discussed later) for interactive image display and manipulation. Linux is the platform of choice. BMRT (Blue Moon Rendering Tools) is also used to render production quality images (refer to section 4.4). Both the software and operating system in use are freely available, and are mostly open source.

4.1 File Formats

To aid further discussion in this chapter, it is useful to know what file types are supported by the system. The system is able to open and read from OFF object files. The system also supports three other file types that have been especially created for it. The three file types are scene files, intermediate object files and test file objects files, each will be explained in more detail.

- Scene File (.scf) – this file is used to store sphere information, and is later used to draw the spheres in the holodeck. Each line is made up of sphere information in the following way: x y z radius r g b. The following is an extract of a scf file.

```
 0.000000 0.000000 0.000000 0.050000 0.000000 0.000000
0.000000
-1.000000 0.000000 0.000000 0.050000 0.000000 0.000000
0.000000
-0.950000 0.000000 0.000000 0.050000 0.000000 0.000000
0.000000
-0.900000 (                              000 0.000000
0.000000
```

Extract 1. Example of scf file

- Intermediate Object File (iof) – this file is used to store the information retrieved from the OFF files. Each line contains triangle coordinates and triangle normals in this format: x y z normal_x normal_y normal_z

```
0.122916 -0.517813 -0.025671   0.684301 0.239955  0.688589
0.148788 -0.517813 -0.076252   0.907180 0.236300 -0.348120
0.122470 -0.400259 -0.084252   0.961109 0.172053 -0.216024
0.122916 -0.517813 -0.025671   0.684301 0.239955  0.688589
```

Extract 2. Example of iof file

- Test File Object (tfo) – this file is used to store a list of primitives that the user wants rendered in the holodeck. The file has the following format: line one contains the number of primitives to be rendered, the lines thereafter alternate between these options: 1. number of vertices, type of primitive, fill of primitive, 2. coordinates in the form x y z.

```
3
1 f f
0 0 0
2 f f
-1 0 0
0 0 0
3 t f
```

Extract 3. Example of tfo file

4.2 Shading

Shading is performed by three main functions within the 'CreateSceneFile' class. The algorithms associated with shading may be found in Section 3.3 Shading. Note will be made here of how these functions were implemented. The C++ prototypes are listed below, and will be used for explanation purposes. The full method source code may be viewed in Appendix A.

```
void CreateSceneFile::DrawPointPhong (Point n1, Point n2, Point n3, double x, double y,
                                      double z);
```

```
void CreateSceneFile::GetPointWeights (Point * points, double x, double y, double z);
```

```
double CreateSceneFile::intensity (double lx, double ly, double lz, double nx, double ny,
                                   double nz, double matambient, double matdiffuse, double
                                   matspecular, double matexponent);

void CreateSceneFile::calculateLighting (double nx, double ny, double nz,
                                         double &r, double &g, double &b);
```

The triangle normals required by ‘DrawPointPhong’ are calculated when the object data file information is extracted from an object file. The coordinates for each triangle, as well as its normals are written to an .iof file for extraction within the ‘CreateSceneFile’ class.

Once the ‘CreateSceneFile’ class has read a triangle and it’s normals from the .iof file, shading is done for that particular triangle. The class will only read the information for the next triangle once it is finished dealing with the current triangle.

To begin the shading process, the GetPointWeights method is called, with the array of points containing the three vertices of the current triangle, and x, y and z are the coordinates of the sphere to be shaded. Using the formulae outlined in section 3.3, the weights (stored in a global array) are calculated for the sphere.

At this point it is necessary to check for division by zero errors that might occur. If not caught, these errors cause undefined values to be passed through the shading functions. These undefined values cause blank spaces within the image, but the program cannot render a sphere whose colour value doesn’t exist. To prevent division by zero errors, checks need to be performed. If the denominator of an equation (refer to section 3.3) is 0, the value that is being calculated, eg w_z, is set to 0, and the calculations proceed. The reason the value is being set to zero is so that the other values (ie w_y and w_x) may still be calculated to fit in with our assumptions (refer to section 3.3). If we set the value to 1, each of the weights would end up being one or close to it, which means that each point has full influence on the weight of the sphere, this cannot be correct.

Once the weights have been calculated, the DrawPointPhong method is called. Points 1, 2 and 3 are once again the triangle vertices, n1, n2 and n3 are the normals read from the .iof file. The method has to calculate the normals of the current sphere, making use of its position and the weights calculated for it. These normals are used by the intensity method to calculate the shading for the sphere.

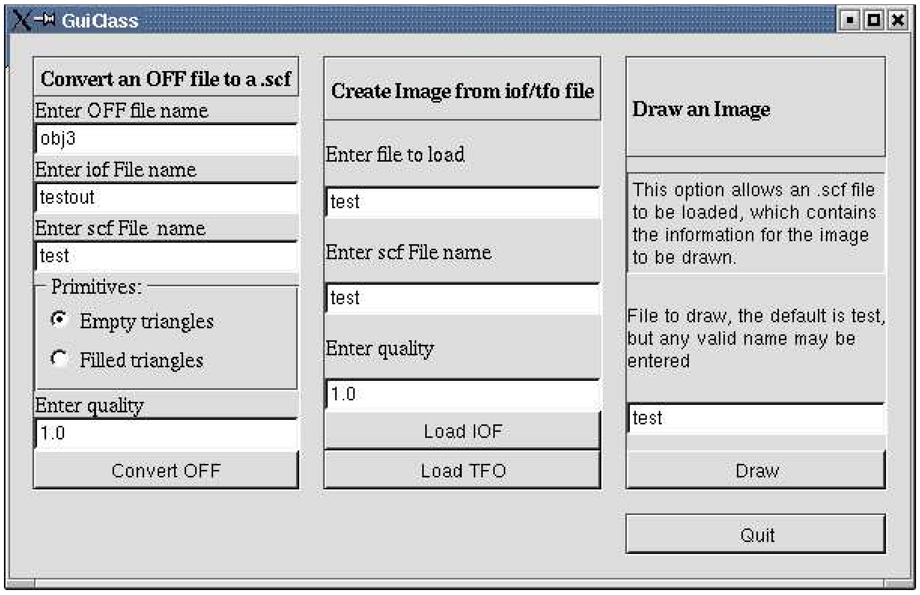
The first three values in this method call are for use in the diffuse calculation. The normals, are for use in the specular and diffuse calculations, as well as for use in calculating the length of the normals. The last four values are the material reflectivities used in the lighting equation (refer to equation 4).

Occassionally a value will have the value of ‘nan’ (not a number). This may be caused by undefined values used in calculations or division by zero. To ensure this does not happen, the isnan (variable) method from the Math.h class is implemented. If the intensity is a ‘nan’, the intensity is set to 0.

4.3 GUIs

4.3.1 The System GUI

As mentioned in section 3.4, a GUI is used to control the object conversion process. Qt is the package used to create all the GUIs within the system. The main GUI integrates the functionality of all the classes, and places the user in control of what is happening.


GuiClass

Convert an OFF file to a .scf

Enter OFF file name: obj3

Enter iof File name: testout

Enter scf File name: test

Primitives: [✓ Empty triangles] Filled triangles

Enter quality: 1.0

Create Image from iof/tfo file

Enter file to load: test

Enter scf File name: test

Enter quality: 1.0

Draw an Image

This option allows an .scf file to be loaded, which contains the information for the image to be drawn.

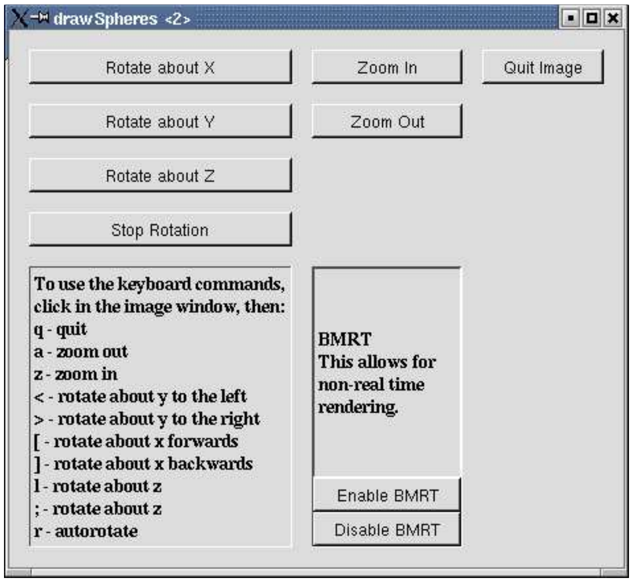
File to draw, the default is test, but any valid name may be entered: test

Figure 10. The system's main GUI

The functionalities of both the CreateSceneFile class and the OFFConverter class are incorporated here. An instance of each class is created each time the GUI program is loaded (header files that contain method prototypes for the classes listed above need to be included). The instance allows for access to the methods available within each class (the header file sources are available in Appendix A). The DrawSpheres class is compiled so that it is a standalone executable. The far right GUI segment in Figure 10, allows the user to run the DrawSpheres program, but it is possible to run it outside of the GUI. The code to call the DrawSpheres executable looks for the exectuable within the current directory.

4.3.2 The DrawSpheres GUI

The DrawSpheres executable contains its' own GUI. Upon execution of this program, two windows (Qt widgets) appear, one contains the image that is being displayed, while the other is the control GUI. To rotate the image, or move in or out (zoom), the user may use the GUI controls, or use the keyboard commands that are also available.



To use the keyboard commands, click in the image window, then:
q - quit
a - zoom out
z - zoom in
< - rotate about y to the left
> - rotate about y to the right
[- rotate about x forwards
] - rotate about x backwards
l - rotate about z
; - rotate about z
r - autorotate

BMRT
This allows for non-real time rendering.

Figure 11. The drawSpheres control GUI

4.4 Rendering

As mentioned in the introduction of this chapter, openGL is used for rendering images, unless the user chooses to use RenderMan to render the images. Part of the objectives for the project include being able to render images using a package such as openGL, as well as being able to use a ray-tracing package such as RenderMan.

To allow for this, the system is designed to be able to switch between the two packages. The option is left entirely to the user. The default renderer is openGL, but the user may switch to RenderMan using the BMRT controls that may be seen in Figure 11. The user may use openGL to manipulate the image until it is to their satisfaction, whereupon they may enable RenderMan to render what they can see in openGL. To discontinue using RenderMan, they simply have to disable it.

Chapter 5

In this chapter we describe the results obtained in this project. Several experiments are run to evaluate how well the system works; the same test object (an OFF palm tree) is in use for each experiment. The first experiment involves measuring the frame rate against a change in quality for the test image. The experiment is run for both filled triangles and frame triangles on the same test object. The second experiment involves measuring the frame rate against a change in sphere radius size. Once again, the experiment is run twice. The results for experiments one and two are in the form of graphs. The third experiment is an evaluation of image quality. An ordinary OFF rendered image will be the standard to which we compare the same image rendered in our system. Several other experiments will compare and evaluate images from openGL with images from BMRT. Finally, we discuss some general observations that arise out of these experiments.

The first two experiments were run on a dual Intel Pentium III 500MHz PC with an Nvidia GeForce graphics card. The third experiment was run on a Celeron 500MHz PC with a Voodoo II Banshee graphics card.

5.1 Quality

This experiment is set up to measure the change in time taken to render a frame when the quality value is increased. To time the frame rate per second, a timing function is added to the DrawSpheres program, which returns the start time and end time (time in seconds) for each frame.

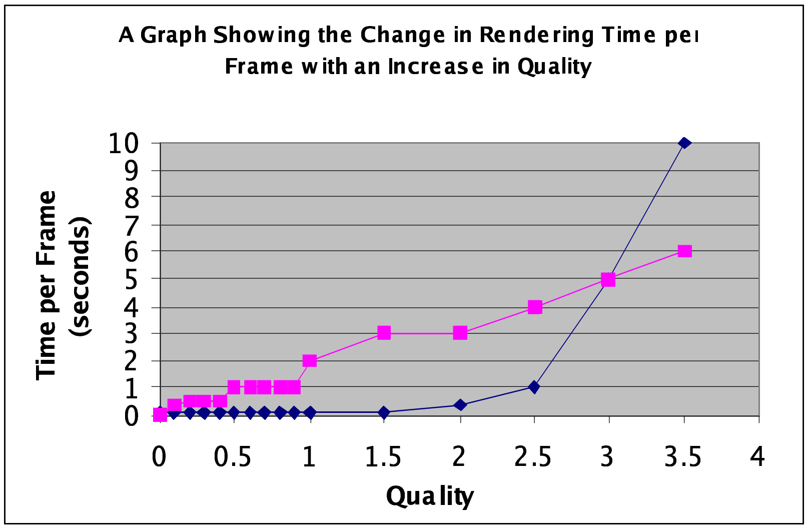
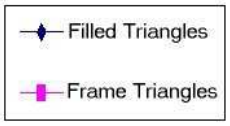


Figure 12. Time per Frame against Quality

The legend for the graphs:



As mentioned previously, one of the purposes of having a quality factor is to give the user control over the rendering time of an image, at the cost of image quality. The graph above shows the results for both the filled triangles and frame triangles.

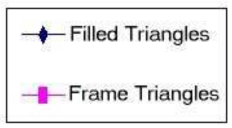
It can be seen that using frame triangles results in a slow rendering time per frame. This is due largely to how many spheres need to be used to draw the triangle, the more spheres the longer it takes to render. When the quality value is 0.8 or above, the time per frame increases by several seconds.

Considering that filled triangles are supposed to be the opposite of frame triangles (in the sense that filled triangles have spheres on the face of the triangle, as opposed to along the edges), it may be somewhat surprising to note that the filled triangles such a high frame rate. The reason for this is the way the numbers of spheres for the filled triangles are calculated, refer to section 3.2.2.2. Initially, little time is taken to render frames, whereas once quality is 1.5 or greater, the time taken increases, although not as drastically as the time taken for frame triangles.

5.2 Radius

Similar to the quality experiment, the radius experiment is set up to measure the change in time taken to render a frame when the quality value is increased. To time the frame rate per second, the same timing function, which returns the start time and end time (time in seconds) for each frame, is used in this experiment.

The legend for the graphs:



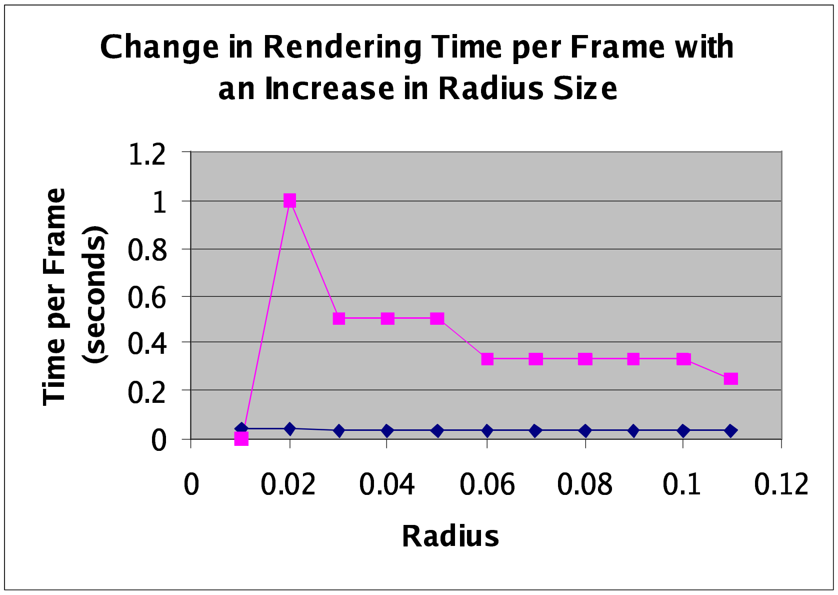


Figure 13. Time per Frame against Radius

Considering that the smallest component in our system is a sphere, it is useful to know how a change in radius affects the rendering rate of the system. The experiments once again make use of filled triangles and frame triangles; the quality for the image is set to one.

The frame triangles frame rate increases with an increase in radius size, which is expected. The reason for this may be seen in the equation that is needed to draw lines out of spheres (a frame triangle is made up of three lines).

*Number of spheres = quality * (distance/radius size) (5)*

The bigger the radius, the fewer the number of spheres required to draw a line. If there are fewer spheres, the image can render faster as there are less objects to draw.

The filled triangles frame rate increases initially and then plateaus. Radius is used to calculate the sphere area, which is used to calculate the number of recursions. This is right considering that the system does not need to use the sphere area again except for when it records the sphere information; it is dependent on how many recursions there are. While rendering, the system has to draw the spheres, and although they are getting fewer in number, they are larger spheres.

While the radius is still relatively small there are no ill effects to be seen in the image. The problem arises then when the radius becomes larger. Although the quality may still be high, the image itself will not be a good representation of what the image should look like. An example may be seen in Figure 14 below. Figure 14 is a BMRT rendered image of a palm tree, using filled triangles and a radius size of 0.11.

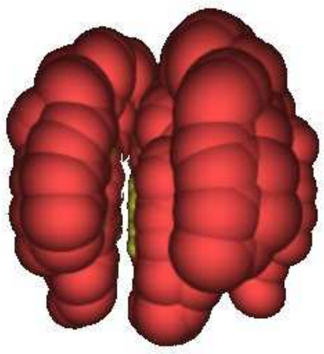


Figure 14. A BMRT image of a palm tree

5.3 Image Evaluation

The first image evaluation experiment involves rendering an OFF object the standard way (standard image), and comparing the result with the same object rendered in our system (test image). The test image has been rendered at a quality level of three (high image quality), using filled triangles (this is so that the image does not have gaps in it, ie so we get the appearance of a solid image, which may be compared to the standard image with appears solid).

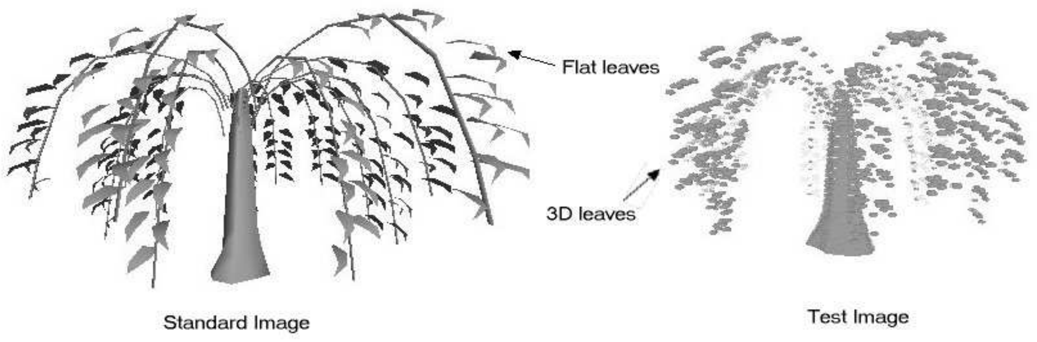


Figure 15. Palm tree rendered using two different techniques

One of the first differences to note between the two images is the clarity of the standard image. Edges to the leaves and trunk may be clearly seen, whereas in the test image the edges are not so well defined. The test image has an almost ‘bumpy’ appearance because of the spheres. It is quite difficult to approximate straight lines with spheres, to do so, one would have to set the quality value very high, which would result in long rendering times. Looking at the test image though, it is still possible to get quite a good approximation of what the image looks like.

Another difference is the appearance of the leaves in both images. The standard images’ leaves appear to be flat, and look almost 2D. The test image, however, has leaves that look 3D, this is largely due to the nature of the spheres. Spheres are inherently 3D, and placing spheres together results in a 3D look.

The second image evaluation experiment shows the difference between the same object rendered using filled triangles, and frame triangles.

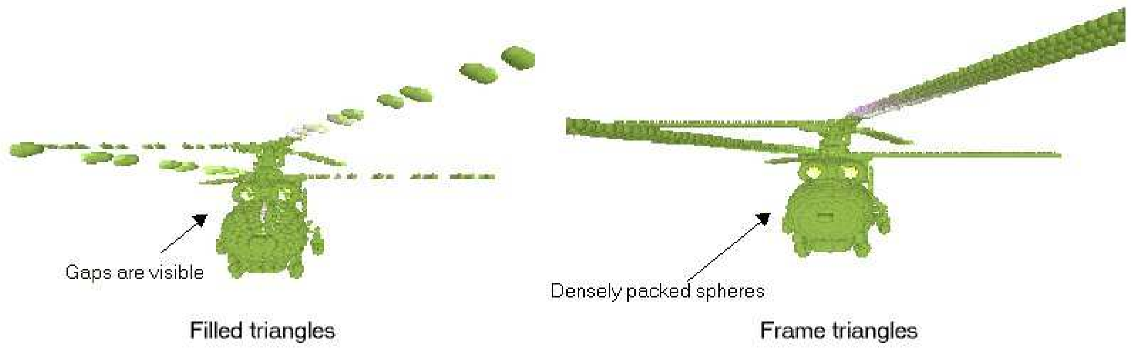


Figure 16. Helicopters rendered using different primitives

In this situation, the frame triangle image looks of higher quality, although for the experiment, a quality level of 1 is used. To obtain the same image quality seen in the frame triangle image, the filled triangle image would need to be set to quality 3 or 4. As can be seen, shading works for either instance, the exact same effects may be seen in each image.

The third image evaluation experiment shows the difference between the same object rendered using RenderMan and OpenGL.

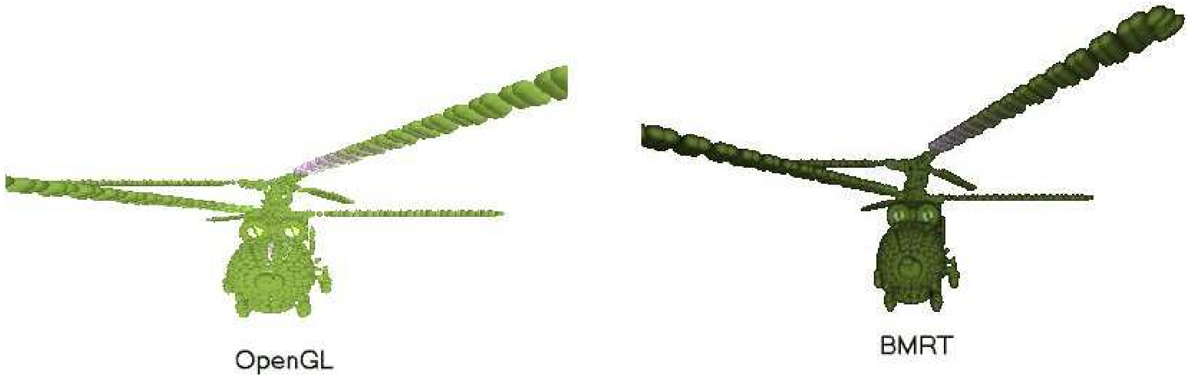


Figure 17. Helicopters rendered using OpenGL and BMRT

The image quality between the two images is vastly different; the quality value for both images is set to 1. The OpenGL image appears saturated by the colours, whereas the BMRT image appears darker. Once again, the shading effects can be clearly seen in both images. The BMRT image takes a lot longer to render. OpenGL is used for ‘real-time’ image movement and transformation, whereas BMRT is used just to get a high quality image.

Chapter 6

In this project we set out to create a virtual holographic technology, and evaluate how it performs. We have successfully implemented a working 3D technology that can be used on any standard computer. The images produced by our system are both unique and effective in the sense that it is clearly visible what the image is meant to be.

Creating images out of spheres is innovative, but is not always ideal to use. Sometimes, clearly defined images such as the standard image in Figure 15 are required; in which case using our system would not yield the desired result. Another aspect that needs to be considered is how much time is a user willing to spend rendering images. Depending on quality, radius and computer speed, rendering can take a long time. This is because of the time required for calculations, as well as reading from the scene file for every frame.

An advantage of our system is the inherent 3D of the images. Even if one renders a triangle, it will appear 3D because of the spheres used to create it. Another advantage is the range of primitives available in the system. Even though all the primitives were not used, they are still available and can be combined to create interesting 3D objects. The tetrahedron is a prime example, this is the first time a tetrahedron has been created solely out of spheres, and the uses for it are endless. Tetrahedrons are 3D primitives; they can be used to create 3D images, just as triangles are used to create 2D images.

The shading in our system is another interesting advancement. This is one of very few 3D shaders available, and it is definitely the only one of its kind. Most shading techniques rely on per-pixel shading, as opposed to our approach that does it on a per-sphere basis.

The ability to switch between renderers is also quite useful. It is convenient to be able to see an approximation image before rendering it. This allows the user to get the image right the first time, as opposed to having to re-render it several times because they cannot see what it is that is being rendered.

Thus we conclude that we have achieved what we set out to accomplish. Our system works, and though not always as fast as ordinary rendering systems, it allows for variety. There is no longer a need to just create images out of pixels, they can now be created using spheres as well.

References:

[Ale, 98]	Alexander, D. A Brief Biography of Gene Roddenberry. http://www.roddenberry.com/creations/bio/groddenberry.bio.html, 1998.
[Ban, 02]	Bangay, S. Computer Graphics Course Notes 2002. http://csshaun.ict.ru.ac.za/graphicsnotes/, 2002.
[Bel, 94]	Bell, J. Holodeck and Computers FAQ. http://www.nwnet.co.uk/gwright/holodeck.htm, 1994.
[Ber, 02]	Berger, M. 3D Display Offers a New View. http://cssvc.pcworld.compuserve.com/ computing/cis/article/0,aid,102372,00.asp, 2002.
[Bon, 02]	Bonsor, K. How Holographic Environments Will Work. http://www.howstuffworks.com/holographic-environment.htm/printable, 1998-2002.
[Dan et al., 99]	Daniilidis, K., Mulligan, J., McKendall, R., Majumder, A., Kamberova, G., Schmid, D., Bajcsy, R., Fuchs, H. Towards the Holodeck: An Initial Testbed for Real-Time 3D-Teleimmersion. http://www.cs.unc.edu/~majumder/PAPER/upenn.html, 1999.
[Din, 02]	Dingliana, J. CS5 Course Notes: Computer Graphics Hardware. http://isg.cs.tcd.ie/dingliaj/cs5/notes/CS5_2Hardware.ppt, 2002.
[Dun, 99]	Dunn, P. Cybersphere Brings Star Trek' s Holodeck Closer to Reality http://www.warwick.ac.uk/news/pr/230, 1999.

[Fol et al., 01]	Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F. Computer Graphics Principles and Practice. California: Addison-Wesley Publishing Company, 2001.
[Fol et al., 84]	Foley, J.D. & Van Dam, A. Fundamentals of Interactive Computer Graphics. California: Addison-Wesley Publishing Company, 1984.
[Fre, 02]	Freedman, D. H. Holograms in Motion. http://www.technologyreview.com/articles/ print_version/freedman1102.asp, 2002.
[Hal, 97]	Halle, M. Autostereoscopic displays and computer graphics. http://web.media.mit.edu/~halazar/autostereo/autostereo.html, 1997.
[Ham, 99]	Hammersley, T. Fast Phong Shading. http://www.whisqu.se/per/docs/graphics8.htm, 1999.
[Hea et al. 86]	Hearn, D. & Baker, M.P. Computer Graphics. New Jersey: Prentice-Hall, 1986.
[Hil, 90]	Hill, F.S. Computer Graphics. New York: Macmillan Publishing Company, 1990.
[How, 02]	HowStuffWorks.com, How 3-D Graphics Work. http://www.howstuffworks.com/3dgraphics.htm/printable, 1998-2002.
[Jac, 01]	Jacobs, S. Making the holodeck a reality. http://electronics.cnet.com/electronics/0-1577332-7-5700991.html, 2001.
[Kra, 95]	Krauss, L. The Physics of Star Trek. South Carolina: Flamingo Publishing, 1995.
[Kuj et al., 99]	Kujawinska, M., Pawlowski, M., Sutkowski, M., Sitnik, R., Franke, N., Kniec, M. How Does Holography Work?. http://holo.mchtr.pw.edu.pl/html/holography.html, 1999.
[Las, 02]	Laser Magic Productions, The TranScreen and The HoloTank. http://www.laser-magic.com/transscreen.html and http://www.laser-magic.com/holotank.html, 2002.
[Lev, 02]	Levoy, M. About Gamma Correction. http://graphics.stanford.edu/gamma.html, 1994-2002.
[Mic, 94]	Michael, J. Fractal Shape Changing Robots. http://easyweb.easynet.co.uk/~robodyne/stellar/holodeck.htm, 1994-1997.
[Nas, 02]	Nasa, Making Three-Dimensional Holograms Visible From All Sides. http://www.nasatech.com/Briefs/Apr02/NPO20101.html, 2002.

[Out et al., 99]	Outwater, C. & Hamersveld, V. Practical Holography. http://www.holo.com/holo/book/book1.html, 1995-1999.
[Rad, 01]	Rademacher, P. Ray Tracing: Graphics for the Masses. http://www.acm.org/crossroads/xrds3-4/raytracing.html, 2001.
[Sal, 01]	Salvator, D. ExtremeTech 3D Pipeline Tutorial. http://www.extremetech.com/print_article/0,3998,a=2674,00.asp, 2001.
[Swa et al., 01]	Swartout, W., Hill, R., Gratch, J., Johnson, W. L., Kyriakakis, C., LaBore, C., Lindheim, R., Marsella, S., Miraglia, D., Moore, B., Morie, J., Rickel, J., Thi Ú baux, M., Tuch, L., Whitney, R., and Douglas J. Toward the Holodeck: Integrating Graphics, Sound, Character and Story. New York: ACM Press, In Proceedings of Fifth International Conference on Autonomous Agents, pp. 409-416, May 2001.
[Wat, 89]	Watt, A. Fundamentals of Three-Dimensional Computer Graphics. London: Addison-Wesley Publishing Company, 1989.
[Web, 02]	Webopedia.com frame buffer. http://www.webopedia.com/TERM/F/frame_buffer.html, 2002.

Appendix A

This appendix contains the source code for the algorithms used within the main thesis.

```
void
CreateSceneFile::drawLine (Point * points, Point start, Point end,
                        Point n1, Point n2, Point n3, const double quality)
{
  Point centerSphere;          // to be used when drawing single spheres

  double radius = Radius_Size;        //set up a local radius variable

  double temp = 0.0;           // initial value



  double x = start.coordinates[0];
  double y = start.coordinates[1];
  double z = start.coordinates[2];

  double dx = calcChange (end.coordinates[0], start.coordinates[0]);

  double dy = calcChange (end.coordinates[1], start.coordinates[1]);

  double dz = calcChange (end.coordinates[2], start.coordinates[2]);



  double distance = distanceCalculation (start, end, dx, dy, dz);
  double steps = stepCalculation (distance, radius, quality);

  for (double i = 0; i < steps; i++)
    {
      GetPointWeights(points, x, y, z);
      DrawPointPhong(n1, n2, n3, x, y, z);

      singleSphere (x, y, z, radius);

      x += calcNewCoords (x, dx, steps);        // x coordinate
      y += calcNewCoords (y, dy, steps);        // y coordinate
      z += calcNewCoords (z, dz, steps);        // z coordinate
    }

}

void
CreateSceneFile::filledTriangle (Point point1, Point point2, Point point3,
                                 Point n1, Point n2, Point n3,
                                 Point * points, const double quality)
{
  int numRec = calculateRecursions (point1, point2, point3, quality);
  compareAreas (point1, point2, point3, n1, n2, n3, points, quality, numRec);
}

int
CreateSceneFile::calculateRecursions (Point P1, Point P2, Point P3,
                                      const double quality)
{
  double sphereArea = calculateSphereArea ();
  double triangleArea = calculateTriangleArea (P1, P2, P3);
  int numRecursions =
    int (quality * log10 (triangleArea / sphereArea) / log10 (4));

  return numRecursions;
}

void
CreateSceneFile::compareAreas (Point P1, Point P2, Point P3,
                               Point n1, Point n2, Point n3,
                               Point * points, double quality, int numRecs = 0)
{

  if (numRecs > 0)
    {
      numRecs--;
      //declare new corner points for the smaller triangle
      Point newPoint1;
      Point newPoint2;
      Point newPoint3;

      newPoint1.coordinates[0] = (P1.coordinates[0] + P2.coordinates[0]) / 2;
      newPoint1.coordinates[1] = (P1.coordinates[1] + P2.coordinates[1]) / 2;
      newPoint1.coordinates[2] = (P1.coordinates[2] + P2.coordinates[2]) / 2;

      newPoint2.coordinates[0] = (P1.coordinates[0] + P3.coordinates[0]) / 2;
      newPoint2.coordinates[1] = (P1.coordinates[1] + P3.coordinates[1]) / 2;
      newPoint2.coordinates[2] = (P1.coordinates[2] + P3.coordinates[2]) / 2;

      newPoint3.coordinates[0] = (P3.coordinates[0] + P2.coordinates[0]) / 2;
      newPoint3.coordinates[1] = (P3.coordinates[1] + P2.coordinates[1]) / 2;
      newPoint3.coordinates[2] = (P3.coordinates[2] + P2.coordinates[2]) / 2;

      compareAreas (P1, newPoint1, newPoint2, n1, n2, n3, points, quality, numRecs);
      compareAreas (newPoint1, newPoint2, newPoint3, n1, n2, n3, points, quality,
                    numRecs);
      compareAreas (newPoint1, P2, newPoint3, n1, n2, n3, points, quality, numRecs);
      compareAreas (newPoint2, newPoint3, P3, n1, n2, n3, points, quality, numRecs);
    }
  else
    {
      Point centerPoint;

      centerPoint.coordinates[0] =
          (P1.coordinates[0] + P2.coordinates[0] + P3.coordinates[0]) / 3;
      centerPoint.coordinates[1] =
          (P1.coordinates[1] + P2.coordinates[1] + P3.coordinates[1]) / 3;
      centerPoint.coordinates[2] =
          (P1.coordinates[2] + P2.coordinates[2] + P3.coordinates[2]) / 3;

      GetPointWeights (points, centerPoint.coordinates[0],
                       centerPoint.coordinates[1],
                       centerPoint.coordinates[2]);
      DrawPointPhong (n1, n2, n3, centerPoint.coordinates[0],
                      centerPoint.coordinates[1], centerPoint.coordinates[2]);

    }
}

void
CreateSceneFile::solidTetrahedron (Point point1, Point point2, Point point3,
                                   Point point4, Point * points,
                                          const double quality)
{
  Vector a = findVector (point1, point2);
  Vector b = findVector (point1, point3);
  Vector c = findVector (point1, point4);
  Vector crossBC = crossProduct (b, c);

  double dot = dotProduct (a, crossBC);
  double tetrahedronVolume = (1.0 / 6.0) * fabs (dot);
  double newRadius = cbrt ((tetrahedronVolume / 2.3) / Pi);

  if (newRadius > quality * Radius_Size)
    {
      Point p1Top2;
      Point p1Top3;
      Point p1Top4;
      Point p2Top3;
      Point p2Top4;
      Point p3Top4;

      //Halfway between points 1 and 2
      p1Top2.coordinates[0] =
            (point1.coordinates[0] + point2.coordinates[0]) / 2;
      p1Top2.coordinates[1] =
            (point1.coordinates[1] + point2.coordinates[1]) / 2;
      p1Top2.coordinates[2] =
            (point1.coordinates[2] + point2.coordinates[2]) / 2;

      //Halfway between points 1 and 3
      p1Top3.coordinates[0] =
            (point1.coordinates[0] + point3.coordinates[0]) / 2;
```

```
p1Top3.coordinates[1] =
    (point1.coordinates[1] + point3.coordinates[1]) / 2;
p1Top3.coordinates[2] =
    (point1.coordinates[2] + point3.coordinates[2]) / 2;

//Halfway between points 1 and 4
p1Top4.coordinates[0] =
    (point1.coordinates[0] + point4.coordinates[0]) / 2;
p1Top4.coordinates[1] =
    (point1.coordinates[1] + point4.coordinates[1]) / 2;
p1Top4.coordinates[2] =
    (point1.coordinates[2] + point4.coordinates[2]) / 2;

//Halfway between points 2 and 3
p2Top3.coordinates[0] =
    (point2.coordinates[0] + point3.coordinates[0]) / 2;
p2Top3.coordinates[1] =
    (point2.coordinates[1] + point3.coordinates[1]) / 2;
p2Top3.coordinates[2] =
    (point2.coordinates[2] + point3.coordinates[2]) / 2;

//Halfway between points 2 and 4
p2Top4.coordinates[0] =
    (point2.coordinates[0] + point4.coordinates[0]) / 2;
p2Top4.coordinates[1] =
    (point2.coordinates[1] + point4.coordinates[1]) / 2;
p2Top4.coordinates[2] =
    (point2.coordinates[2] + point4.coordinates[2]) / 2;

//Halfway between points 3 and 4
p3Top4.coordinates[0] =
      (point3.coordinates[0] + point4.coordinates[0]) / 2;
  p3Top4.coordinates[1] =
      (point3.coordinates[1] + point4.coordinates[1]) / 2;
  p3Top4.coordinates[2] =
      (point3.coordinates[2] + point4.coordinates[2]) / 2;

  //Start getting smaller Tetrahedrons

  Point centerSphere1;

  centerSphere1.coordinates[0] =
      (p1Top2.coordinates[0] + p2Top3.coordinates[0] +
       p1Top4.coordinates[0] + p3Top4.coordinates[0]) / 4;
  centerSphere1.coordinates[1] =
      (p1Top2.coordinates[1] + p2Top3.coordinates[1] +
       p1Top4.coordinates[1] + p3Top4.coordinates[1]) / 4;
  centerSphere1.coordinates[2] =
      (p1Top2.coordinates[2] + p2Top3.coordinates[2] +
       p1Top4.coordinates[2] + p3Top4.coordinates[2]) / 4;

  //Deal with diamond
  solidPyramid (p3Top4, p1Top4, p2Top4, p2Top3, p1Top3, points, quality);
  solidPyramid (p1Top2, p1Top4, p2Top4, p2Top3, p1Top3, points, quality);

  solidTetrahedron (point1, p1Top2, p1Top4, p1Top3, points, quality);
  solidTetrahedron (point2, p1Top2, p2Top3, p2Top4, points, quality);
  solidTetrahedron (point3, p2Top3, p3Top4, p1Top3, points, quality);
  solidTetrahedron (point4, p1Top4, p2Top4, p3Top4, points, quality);

}
```

```
}

/* All lighting code adapted from the original, provided by Prof Bangay */
// Phong style shader.
void
CreateSceneFile::DrawPointPhong (Point n1, Point n2, Point n3, double x, double y, double z)
{
  int xi = (int) x;
  int yi = (int) y;

  double nx;
  double ny;
  double nz;

  // get normal at current vertex.
  nx =
    w[0] * n1.coordinates[0] + w[1] * n2.coordinates[0] +
    w[2] * n3.coordinates[0];
  ny =
    w[0] * n1.coordinates[1] + w[1] * n2.coordinates[1] +
    w[2] * n3.coordinates[1];
  nz =
    w[0] * n1.coordinates[2] + w[1] * n2.coordinates[2] +
    w[2] * n3.coordinates[2];

  // get intensity at current vertex.
  double cr;
  double cg;
  double cb;
  calculateLighting (nx, ny, nz, cr, cg, cb);
  singleSphere (x, y, z, Radius_Size, cr, cg, cb);
```

```
}

//Get the w'  s using:

void
CreateSceneFile::GetPointWeights (Point * points, double x, double y,
                                   double z)
{

  double P0x = points[0].coordinates[0];
  double P0y = points[0].coordinates[1];
  double P0z = points[0].coordinates[2];
  double P1x = points[1].coordinates[0];
  double P1y = points[1].coordinates[1];
  double P1z = points[1].coordinates[2];
  double P2x = points[2].coordinates[0];
  double P2y = points[2].coordinates[1];
  double P2z = points[2].coordinates[2];
  double xp = x;
  double yp = y;
  double zp = z;

  double wx = 0.0;
  double wy = 0.0;
  double wz = 0.0;

  //Calculate w0
  double d = (((P0y * P2x - P2y * P0x) *
           (P0z * P1x - P1z * P0x)) -
             ((P0z * P2x - P2z * P0x) *
              (P0y * P1x - P1y * P0x)));
```

```
if (d == 0.0)
  wz = 0.0;
else
  wz = (P1x * (P0y * P2x - P2y * P0x) - P2x * (P0y * P1x - P1y * P0x)) / d;

d = (P0y * P2x - P2y * P0x);
if (d == 0.0)
  wy = 0.0;
else
  wy = (P2x - (P0z*P2x - P2z*P0x)*wz) / d;

d = P0x;
if (d == 0.0)
  wx = 0.0;
else
  wx = (1 - wy * P0y - wz * P0z) / d;

w[0] = wx * x + wy * y + wz * z;

//Calculate w1
d = (((P0x * P1z - P1x * P0z) *
      (P1y * P2x - P2y * P1x)) -
     ((P0x * P1y - P1x * P0y) *
      (P1z * P2x - P2z * P1x)));

if (d == 0.0)
  wz = 0.0;
else
  wz = ((P1y * P2x - P2y * P1x) * P0x - (P0x * P1y - P1x * P0y) * P2x) / d;

d = (P0x * P1y - P1x * P0y);
```

```
if (d == 0.0)
  wy = 0.0;
else
  wy = (P0x - wz*(P0x*P1z - P1x*P0z)) / d;

d = P1x;
if (d == 0.0)
  wx = 0.0;
else
  wx = (1 - wy * P1y - wz * P1z) / d;

w[1] = wx * x + wy * y + wz * z;

//Calculate w2
d = (((P2y * P0x - P0y * P2x) *
      (P2z * P1x - P1z * P2x)) -
     ((P2z * P0x - P0z * P2x) *
      (P2y * P1x - P1y * P2x)));
if (d == 0.0)
  wz = 0.0;
else
  wz = (P1x * (P2y * P0x - P0y * P2x) - P0x * (P2y * P1x - P1y * P2x)) / d;

d = (P2y * P0x - P0y * P2x);
if (d == 0.0)
  wy = 0.0;
else
  wy = (P0x - (P2z * P0x - P0z * P2x) * wz) / d;

d = P2x;
if (d == 0.0)
```

```
    wx = 0.0;
  else
    wx = (1 - wy * P2y - wz * P2z) / d;

  w[2] = wx * x + wy * y + wz * z;

}

// simple lighting routine
double
CreateSceneFile::intensity (double lx, double ly, double lz, double nx, double ny, double nz,
                                                // basic vectors.
                                             double matambient, double matdiffuse, double matspecular, double
                                             matexponent)  // material reflectivities.
{
  double lenl = sqrt (lx * lx + ly * ly + lz * lz);
  lx /= lenl;
  ly /= lenl;
  lz /= lenl;
  double lenn = sqrt (nx * nx + ny * ny + nz * nz);
  nx /= lenn;
  ny /= lenn;
  nz /= lenn;

  double vx = 0.0;
  double vy = 0.0;
  double vz = -1.0;

  double sx = -(lx + vx);
  double sy = -(lx + vx);
  double sz = -(lx + vx);
double lens = sqrt (sx * sx + sy * sy + sz * sz);
sx /= lens;
sy /= lens;
sz /= lens;

double diffuse = lx * nx + ly * ny + lz * nz;
if (diffuse < 0.0)
  diffuse = 0.0;
if (diffuse > 1.0)
  diffuse = 1.0;

double specular = sx * nx + sy * ny + sz * nz;
if (specular < 0.001)
  specular = 0.001;
if (specular > 1.0)
  specular = 1.0;

double i =
  matambient + matdiffuse * diffuse + matspecular * pow (specular,
                                                        matexponent);
if (isnan (i))
  {
    cout << "len " << lenl << " " << lenn << " " << lens << "\n";
    cout << "n " << nx << " " << ny << " " << nz << "\n";
    cout << "w " << w[0] << " " << w[1] << " " << w[2] << "\n";
    i = 0.0;
  }
if (i < 0.0)
  i = 0.0;
if (i > 1.0)
  i = 1.0;
  return i;
}

void
CreateSceneFile::calculateLighting (double nx, double ny, double nz,
                                    double &r, double &g, double &b)
{
  r = intensity (0.1, 0.2, -1.0, nx, ny, nz, 1.0, 1.0, 1.0, 6.0);
  g = intensity (0.1, 0.2, -1.0, nx, ny, nz, 0.3, 0.9, 0.0, 6.0);
  b = intensity (0.1, 0.2, -1.0, nx, ny, nz, 0.3, 0.0, 0.6, 6.0);
}
```