UTS: Introduction to **Computer Graphics**

Major Assignment

Deinyon Davies ***11688025***
Yiannis Chambers ***11699156***

Table of Contents

Project-Wide Dependencies 0

Three-Dimensional Vector Data Structure <vec3.h> 0

Custom Bitmap Parser <bmpLoader.h> 0

Program Select Menu <programSelectMenu.h> 0

Part A: Icosahedron 1

Introduction: 1

Viewing System: 1

Object Model: 2

Offset Pentagon Algorithm 2

Golden Ratio Algorithm 2

Geodesic Algorithm (Final) 3

Data Structure: 3

Display Strategy 3

Texture Patterns 4

Figure 1.6: Icosahedron Texture Page 5

Texture Mapping 5

Lighting 6

Keyboard Controls 7

Screen Shots: 7

Figure 1.7: Final Render 7

Figure 1.8: Render Excluding Texture & Wireframe 7

Part B: Tycho Brahe Planetarium 8

Introduction: 8

Viewing System: 9

Object Model: 9

Original Cylinder: 9

Revised Cylinder: 10

Final Cylinder Modification 10

Initial Construction of Rectangular Buildings: 10

Rectangle Buildings Revision I: 11

Rectangle Buildings Revision 2: 11

Final Rectangle Building Representation: 11

Data Structure: 12
Display Methods: 12
Initial Rendering: 12
Texture Patterns: 13
Figure 2.4: Revised Cylinder Ring (left) and Cap (right) Textures 14
Texture Mapping: 14
Initial Texture Mapping: 14
Revised Texture Mapping: 15
Lighting: 16
Keyboard Controls 16
Screenshots: 16
Figure 2.7: Final Render With (left), and Without (right) Texture Bindings: 16
Part C: Wavefront OBJ Model Loader 17
Introduction 17
Lighting 17
Viewing Volume 18
Data Structure 18
Texture Patterns 19
Figure 3.3: Two Provided Texture Patterns 19
Screen Shots 19
Figure 3.3: Single-Light Revision 19
Figure 3.4: Multiple Light Source Revision 20
Reflection: 20
Yiannis Chambers 20
Achievements: 20
Difficulties: 20
Lessons: 21
Deinyon Davies 22
Achievements 22
Difficulties 22
Lessons 23
References: 23

Project-Wide Dependencies

Three custom libraries were written to provide functionality to all three components of the Major Assignment. Such dependencies were written in header files that are included in each project's C++ source file where necessary. The implementation of a modular framework substantially reduces duplication of code, and provides a rich codebase to encourage compact and logical design.

All project-wide include files are provided in the 'include' subdirectory of the assignment workspace.

Three-Dimensional Vector Data Structure <vec3.h>

The *vec3* template data structure is used extensively in all three assignment components. The structure provides several utility functions and operator overrides, including: vector **magnitude**, **normalisation**, **cross product**, **dot product**, **scalar multiplication** and **scalar division**. The structure allows for simple vector manipulation and storage with little code redundancy.

The include type-defines a *vec3* of type *float* called *vec3f* for shorthand denotation.

Custom Bitmap Parser <bmpLoader.h>

The bitmap parser provides a means for all three programs to load bitmap textures for storage in OpenGL's native floating-point texel format. The library implements two C++ structures whose contents are read directly from any arbitrary bitmap file using the Standard I/O library's file manipulation functions, including *fopen(), fread()* and *fseek().*

The parser is capable of reading bitmaps with widths that are not divisible by 8.

Program Select Menu <programSelectMenu.h>

The Program Select Menu include is a simple collection of functions which create, and provide a callback function to all project components. The library implements the platform independent *system()* function to start a process on a new thread.

Some experimentation was done with the Windows SDK *WINAPI CreateProcess()* function, though use of this function would add unnecessary overhead and platform dependence that could be avoided with the platform independent *system()* function.

Part A: Icosahedron

Introduction:

The Icosahedron is a platonic solid composed of twenty triangular facets. As demonstrated below, it can be observed that the Icosahedron is formed of two opposing convex pentagons with uniform exterior angles of 72^o (*Figure 1*).or based on the vertices of specifically aligned golden rectangles (*Figure 2*). Each vertex of the polyhedron may be plotted on a geodesic.

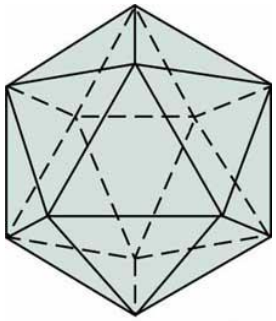


Figure 1.1:
Pentagonal Icosohedron Construction

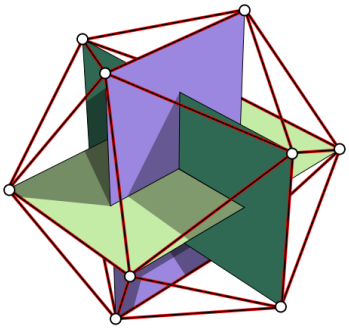


Figure 1.2:
Golden Rectangle Construction

Viewing System:

The viewing system was defined by a call to the OpenGL Utility Toolkit function, gluPerspective(), with a wide field-of-view of 90^o, and a large viewing volume spanning 0.1 to 1,500.0 units into the Z plane. While such a large frustum is not mandatory for drawing geometry that spans no more than 60.0 units into the viewing volume, the distant clipping planes allowed for the drawing of a large line grid.

```
gluPerspective(90.0, width / height, 0.1f, 1500.0f);
```

This call creates a viewing system for the program, modelled as follows:

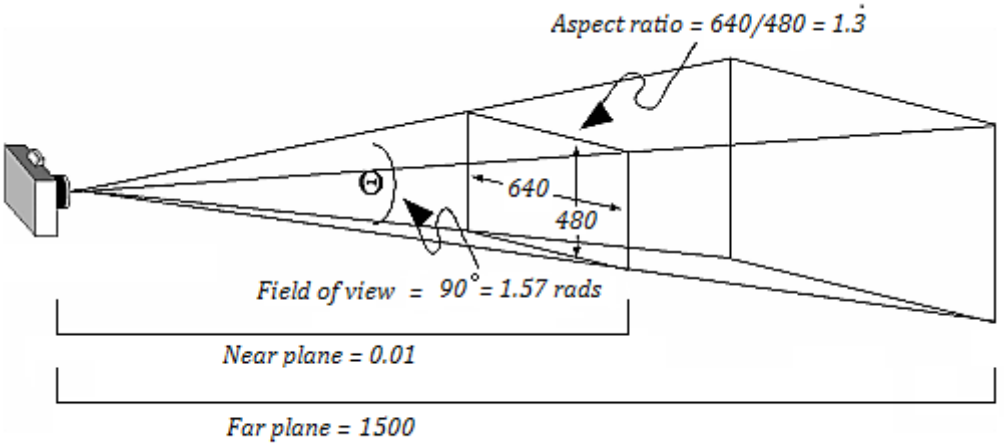


Figure 1.3:
Basic Level Viewing System

Object Model:

Three distinct Icosahedra generation algorithms were prototyped in the development process of Part A, which are detailed in the following subsections:

Offset Pentagon Algorithm

The initial Icosahedron computation algorithm employed the observation that the Icosahedron is composed of two opposing pentagons offset at 180^o from each other, with a distance (d) from the origin defined as:

$$d = \sqrt{c^2 - 2r}$$

Where r is the radius of the Icosahedron and c is defined as:

$$c = \left| \begin{matrix} \mathrm{r}\cos 0 - r\cos 72° \\ r\sin 0 - r\sin 72° \end{matrix} \right|$$

Given that 72^o is the exterior angle of a pentagon.

Each pentagon's five vertex coordinates were computed with the parametric formula:

$$\begin{aligned} x &= r\sin\theta \\ y &= \frac{1}{2}d \qquad 0 \le \theta \le 2\pi \\ z &= r\cos\theta \end{aligned}$$

Where θ increments by $\frac{1}{5}2\pi$, such that the computation results in five vertices along the geodesic.

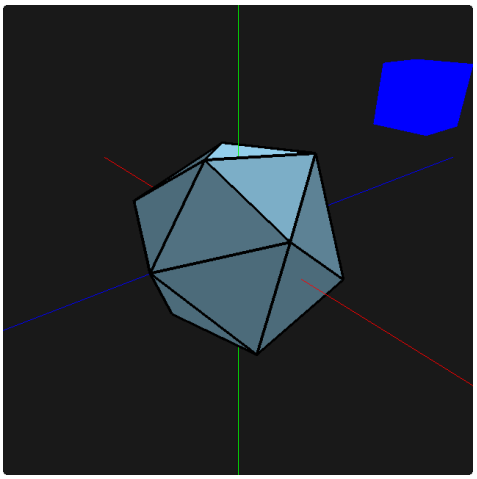


Figure 1.4: Initial Icosahedron Implementation

Golden Ratio Algorithm

An alternative algorithm finds the vertices of the Icosahedron through the application of three orthogonal **golden rectangles,** whose side lengths conform to the golden ratio:

$$\frac{1+\sqrt{5}}{2}$$

Although efficient, the algorithm required facets to be formed of manual, arbitrary links between the three golden rectangles, which resulted in inextensible code.

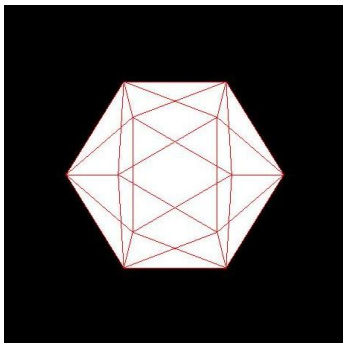


Figure 1.5: Golden Ratio Implementation

Geodesic Algorithm (Final)

The selected vertex coordinate computation algorithm implements two constants: r - the radius of the solid, and φ - the constant pole-to-pole angle at which the five vertices lie, where s is the size of the solid.

$$r = s\sin(\tfrac{2\pi}{5}) \qquad \varphi = 2\sin^{-1}\left(\frac{1}{2\sin\frac{2\pi}{5}}\right)$$

The resulting coordinates obtained by the algorithm are stored in a **data structure** for efficient pre-computed drawing.

Data Structure:

Initially, the data structure took the form of simple one- and two-dimensional arrays.

The Golden Rectangle Algorithm utilized a series of three four-element arrays to store the four vertices of each of the three Golden Rectangles (i.e. [4][3]). The data structure became obsolete with the realisation of the final geodesic algorithm.

The final implementation brought about a portable and modular class implementation, named *Icosahedron*, which stores a two-dimensional array of *vec3f* vertex coordinates, in the form *[2][6]*. The high-order dimension is two elements long, and is used to denote the two sides of the polyhedron. The second dimension holds six *vec3f* data structures: index zero stores the coordinates of the pentagon midpoint vertex of the current side, while the remaining five indices store the five vertex coordinates defining the vertices of a pentagon. Each side effectively composes a **cap**.

Display Strategy

In all implementations, the icosohedron object was displayed using the Open GL primitives GL_TRIANGLES, seen to be the most effective at dealing with the equilateral triangular faces of the icosohedron.

The display method was divided into two major components: constructing the upper and lower pentagons of the icosohedron using the set of calculated points; and joining these two pentagons to form the final model.

Constructing the upper and lower pentagons involved creating a triangle with the centre point of the pentagon, and the current and subsequent point in the data structure, using a custom method that took three vertices (*d_triangle()*). The procedures followed by this method is expressed in the following pseudocode (excluding texture coordinate and face normal allocation):

```
for each side (pentagon of the icosohedron)

{

     for each point in the pentagon
     {

          glBegin(GL_TRIANGLES);

          glVertex3fv(points[side][0]); //the center point
          glVertex3fv(points[side][point]);
          glVertex3fv(points[side][point+1]);

          glEnd();
     }
}
```

Joining the two pentagons followed a similar format; four points were passed to a custom built method (*d_triQuad()*), which operated as expressed below:

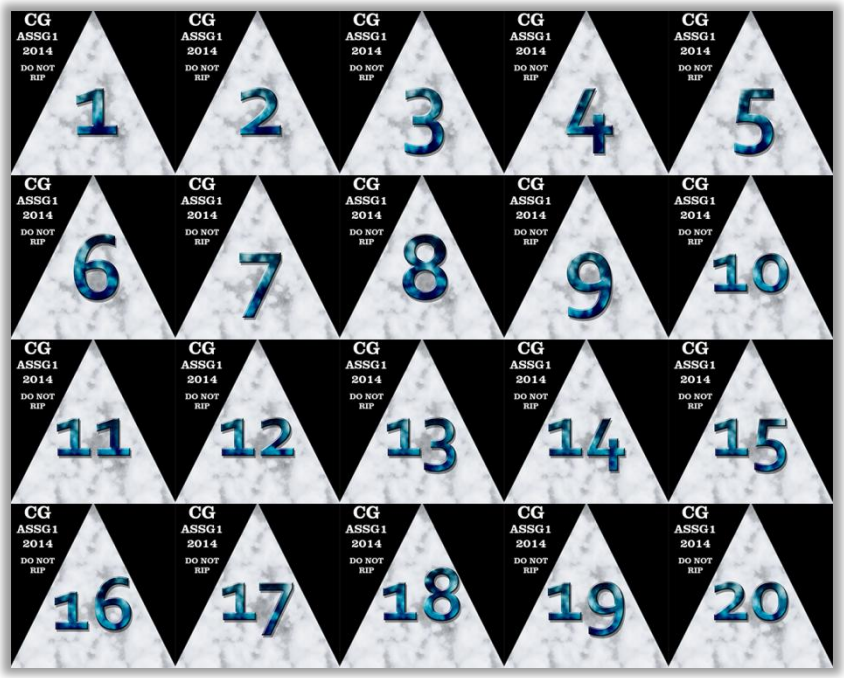
```
glBegin(GL_TRIANGLES);

for each point in the pentagon plus one

{
     glVertex3fv(points[0][(point) % NUM_POINTS]);
     glVertex3fv(points[1][(point+2) % NUM_POINTS]);
     glVertex3fv(points[1][(point+3) % NUM_POINTS]);
     glVertex3fv(points[0][(point+1) % NUM_POINTS]);
}

glEnd();
```

Faces were then culled and depth tested through the enabling of GL_DEPTH_TEST and GL_CULL_FACE.

Texture Patterns

A single texture page of five-by-four uniform textures was implemented. The texture page was created using Adobe Photoshop, and provides a unique identifier for each facet of the Icosahedron.

Figure 1.6: Icosahedron Texture Page



Texture Mapping

The Icosahedron texture mapping algorithm determines the correct texture coordinates for each facet of the polyhedron, from the texture page. A utility function, *get_uv()*, takes an index, along with the number of horizontal and vertical tiles, to infer the correct U/V texture coordinates (as a unit vector) for each face.

A utility function, the aformentioned *d_triangle()*, takes three vectors and a texture index to construct a triangle with the appropriate texture.

The following pseudocode is a revised analysis of the overall function of *d_triangle()*:

```
for each side (pentagon of the icosohedron)
{
     for each point in the pentagon
     {
          glBegin(GL_TRIANGLES);

          calculate a normal vector with points[side][0],
          points[side][point], and points[side][point + 1]

          set the current normal to the one calculated above

          get u and v coordinates for the texture from the
          texture page using the index of the current point
```

```
        glTexCoord2f(x, y);
             glVertex3fv(points[side][0]);
        glTexCoord2f(x + tile width/2, y + tile height);
             glVertex3fv(points[side][point]);
        glTexCoord2f(x + tile width, y);
             glVertex3fv(points[side][point+1]);

        glEnd();
}
```

Lighting

Normal vector computation is performed by the *d_triNorm()* function, and is called per-facet by the aforementioned *d_triangle()* function. The function implements the operator overloads of the *vec3* structure to find a vector orthogonal to the surface. The following pseudocode describes the function of *d_triNorm()*:

```
Isolate triangle edge directions from their spatial
positions by finding the edge delta vectors:

vec3 A = P2 - P1

vec3 B = P3 - P1

Find the vector orthogonal to the surface defined by P1-
P2-P3 by computing the cross product of A and B:

vec3 N = A * B

Normalize the resulting vector so that it is orthonormal
to the surface:

N = N.normalize()

Return N
```

Vertex normals are not interpolated between facets so as to preserve the Icosahedron's distinct edges, though rendering is nonetheless performed using the **Gouraud** method of colour interpolation across vertices.

Two distinct light types exist in the scene, each with ambient, diffuse and specular contributions. Material ambient and diffuse albedos are defined using *glColorf()* by enabling GL_COLOR_MATERIAL for both ambient and diffuse components. The material's specular albedo and specular power is defined by configuring the material's GL_SPECULAR and GL_SHININESS material properties.

The Icosahedron's material has a low specular power of 32.0, so as to produce a soft specular reflection when the viewer aligns with the light's reflection vector.

A directional light (infinite attenuation) is defined in the direction {200.0, 200.0, 100.0}. The light's W component is defined as 0.0 so as to define its position as a direction.

A spot light is declared with a cutoff angle of 130^{o}, and exponent of 32.0. The light has blue diffuse and specular contributions. A position and direction is defined for the spot light so as to point the light toward the Icosahedron from afar.

Keyboard Controls

Camera transformation may be manipulated using the mouse while holding the left mouse button, or with the keyboard controls: 'X', 'Y', and 'Z' to orbit on the X, Y and Z axes respectively.

Screen Shots:

Figure 1.7: Final Render

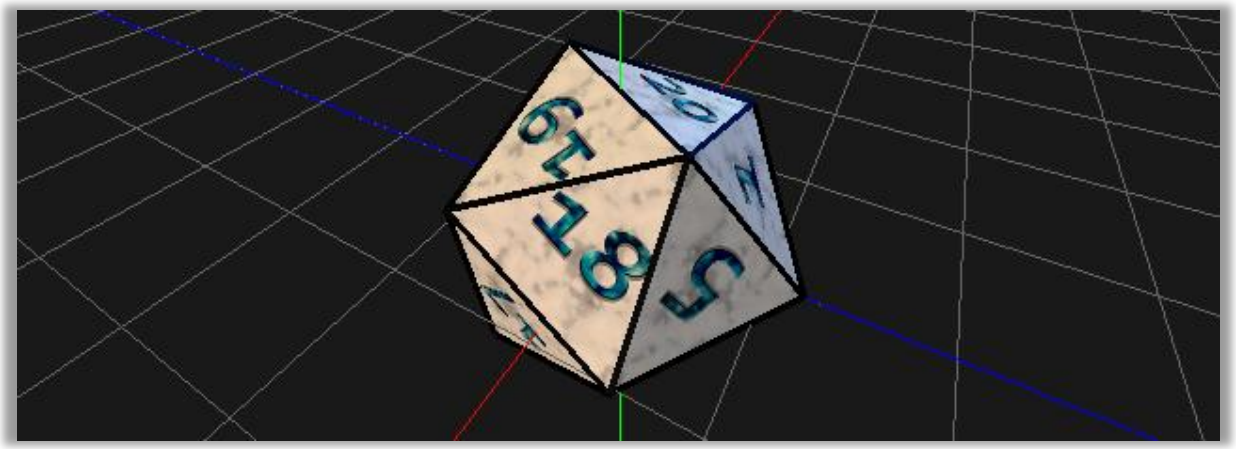
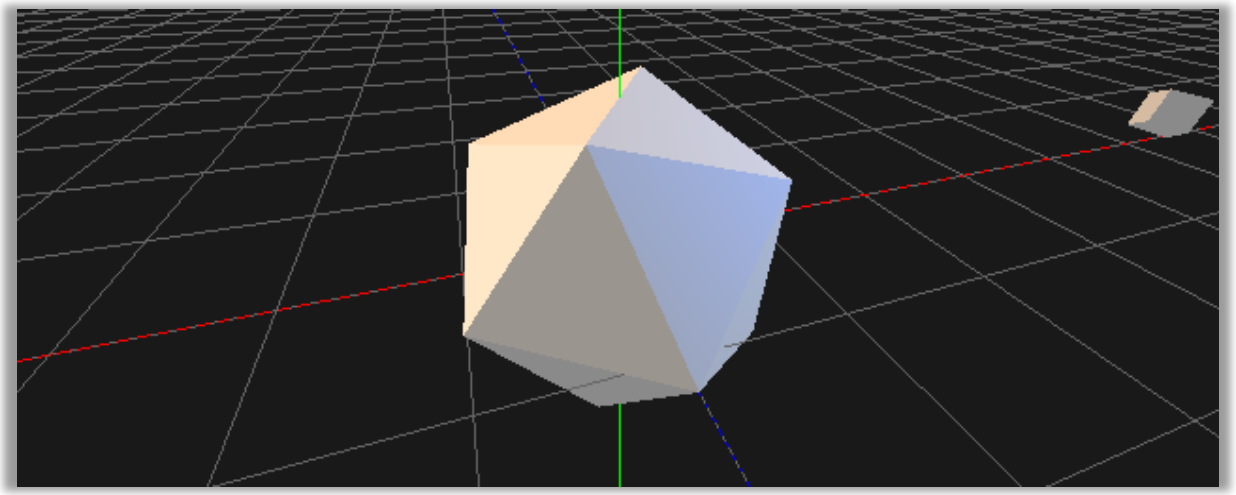


Figure 1.8: Render Excluding Texture & Wireframe



Part B: Tycho Brahe Planetarium

Introduction:

In order to recreate the specified scene of the Tycho Brahe Planetarium, five major components required construction, which became the main goals of the task. These consisted of creating:

- the main cylindrical tower building;
- the five main rectangular buildings in the background;
- the skybox;
- and the lake which separates the Planetarium from the viewer.

The order of construction was tackled in a method of priority; first efforts were directed to setting up the perspective constructing the cylinder. This was achieved in the final version using a set radius and height for the cylinder. From these radius and height values, the cylinder bases were then created, simply by making two cylinders of double the radius of the main cylinder tower with a drastically reduced height. Positions, heights, widths and lengths for the rectangular buildings of the background were derived in the same way, with reference to satellite photography provided by Google Maps. These values were initially passed (once calculated) to a rectangle building class that created a rectangular prism from the set of values. After this, these models were textured using the SOIL library, and lighting put into effect.

In later revisions, a skybox was created, textured with the aforementioned custom bitmap parser library, which encompassed the entire scene. Finally, the lake of water between the viewer and the buildings, a simple function of Y, was developed and implemented in the scene.

Viewing System:

The viewing system was generated via a call to glPerspective/Frustum, noted below with a field-of-view of 70°, and a viewing volume spanning 0.1 to 5000.0 units into the Z plane:

```
gluPerspective(70, 640, 480, 0.1f, 5000.0f);
```

This call creates a viewing system for the program, modelled as follows:

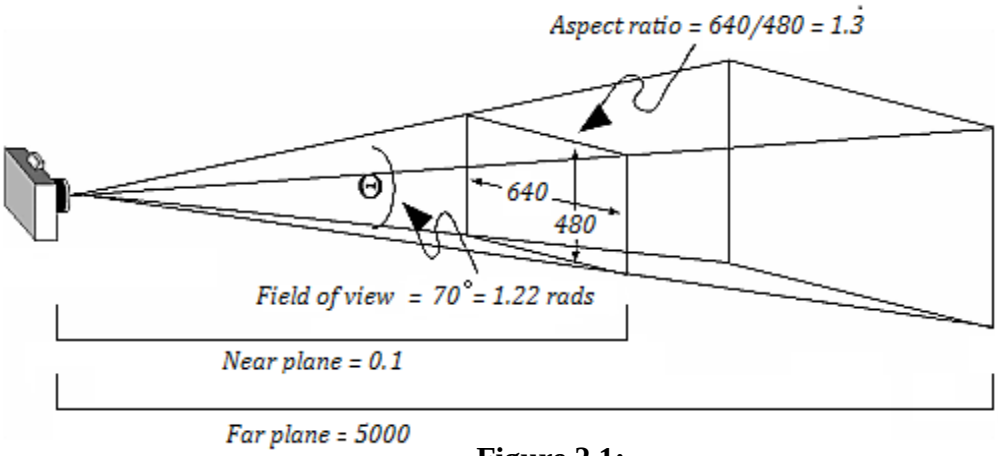


Figure 2.1:
Part B Viewing System

Object Model:

Original Cylinder:

The original design and construction of the cylinder followed standard boundary representation sweeping construction. The cylinder was constructed with 10 cuts to create the polygon mesh of the cylinder, and 20 stacks to define each circle of the sweeping representation.

The x, y and z components of the cylinder were calculated as follows, with a radius (r) of 50, and a height (h) of 100 (with l as an incrementing portion of h defined by the total height multiplied by the fraction of the current cut over the total number of cuts), and where *theta* (θ) is a incrementing variable of a portion of a full rotation in radians (2PI) defined by 2PI multiplied by the fraction of the current stack over the total number of stacks:

$$x = r \times cos(\theta)$$
$$y = h$$
$$z = r \times sin(\theta)$$

$$h = height \times \frac{j}{cuts} \qquad \theta = 2PI \times \frac{i}{stacks}$$

Revised Cylinder:

Due to difficulties with constructing the sloped top face of the cylinder, the construction was revised down to two cuts and 20 stacks, with each set of vertices defining one face.
The bottom portion of the cylinder was constructed as described above; the only modification made was in the construction of the top face of the cylinder.
This entailed the calculation of the y component of each vertex dependant on the current portion of a rotation the vertex was located, as described below:

$$while\ 0 < \theta < PI$$
$$y = r \times sin(\theta)$$

$$while\ PI < \theta < 2PI$$
$$y = r \times sin(-\theta)$$

This calculation produced the angled cylindrical effect required for the main tower of the Tycho Brahe Observatory.

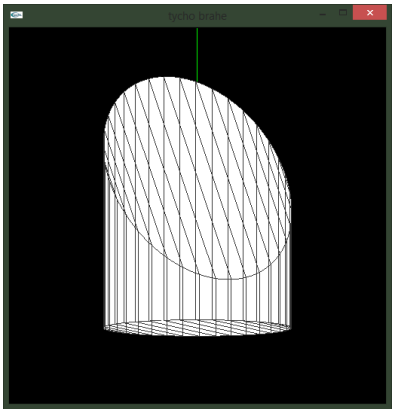


Figure 2.2:
The revised cylinder.

Final Cylinder Modification

The cylinder's slope height (y) was revised to take the form of:

$$y = O_y + h + s\cos\theta$$

Where O_y is the origin's Y component, h is the height of the polyhedron, s is the slope amplitude, and θ is the angle of the current vertex from the origin.

Initial Construction of Rectangular Buildings:

Similar to the initial construction of the cylinder, the rectangular buildings were constructed using standard boundary representation utilizing methods utilized in laboratory work to construct a rectangular prism from a sphere.

In this manner, the coordinates of each vertex were constructed using the incrementing variables *phi* and *theta*, and a static radius (*r*) variable, where *phi* is a proportion of a vertical semi-circular rotation defined in increments of 2 cuts, and *theta* is a proportion of a horizontal circular rotation defined in increments of 4 stacks, as before:

$$x = rsin\varphi\ cos\theta$$

$$y = rsin\varphi sin\theta$$

$$z = r\ cos\varphi$$

$$0 \leq \theta \leq 2\pi \qquad 0 \leq \varphi \leq \pi$$

Rectangle Buildings Revision I:

Due to difficulties with creating rectangles of a variable width and height, the calculation was refined to a sweeping representation, similar to the construction of a cylinder, but sweeping along the x axis rather than the y, and utilizing four stacks instead of 20.

As before, proportional *theta* and *length* variables were utilized in the construction of the objects:

$$x = l$$
$$y = r \times cos(\theta)$$
$$z = r \times sin(\theta)$$

$$l = length \times \frac{j}{cuts} \qquad \theta = 2PI \times \frac{i}{stacks}$$

Rectangle Buildings Revision 2:

The previous sweeping representation also did not meet designer satisfaction in constructing rectangles of a specified width, length and height. Thus, the construction of rectangular buildings was simplified to low level methods; within a class, a set width (*w*), height (*h*) and length (*l*) for each cube, along with centre points (*centreX*, *centreY* and *centreZ*), were defined for each object, and construction of each vertex was specified as follows:

$$v1 = \left\{ centreX - \frac{l}{2}, \quad centreY - \frac{h}{2}, \quad centreZ - \frac{z}{2} \right\}$$
$$v2 = \left\{ centreX - \frac{l}{2}, \quad centreY + \frac{h}{2}, \quad centreZ - \frac{z}{2} \right\}$$
$$v3 = \left\{ centreX + \frac{l}{2}, \quad centreY + \frac{h}{2}, \quad centreZ - \frac{z}{2} \right\}$$
$$v4 = \left\{ centreX + \frac{l}{2}, \quad centreY - \frac{h}{2}, \quad centreZ - \frac{z}{2} \right\}$$
$$v5 = \left\{ centreX - \frac{l}{2}, \quad centreY - \frac{h}{2}, \quad centreZ + \frac{z}{2} \right\}$$
$$v6 = \left\{ centreX - \frac{l}{2}, \quad centreY + \frac{h}{2}, \quad centreZ + \frac{z}{2} \right\}$$
$$v7 = \left\{ centreX + \frac{l}{2}, \quad centreY + \frac{h}{2}, \quad centreZ + \frac{z}{2} \right\}$$
$$v8 = \left\{ centreX + \frac{l}{2}, \quad centreY - \frac{h}{2}, \quad centreZ + \frac{z}{2} \right\}$$

Final Rectangle Building Representation:

Ultimately, the rectangle building construction algorithm was reverted to the initial algorithm, where *r* is multiplied by a scale factor for each component. The algorithm is concise and standardised.

Data Structure:

The data structure for the cylinders and rectangular buildings are simple two dimensional arrays of *vec3f* vectors; the data structure for the cylinder is a two dimensional array that holds the vertices of the two caps of the cylinder, whilst the vertices of the rectangular buildings were stored in much the same way.

Display Methods:

Initial Rendering:

Initial rendering was accomplished, given a set of vertices, using standard OpenGL primitives, namely GL_QUAD_STRIP and GL_POLYGON. The vertices for the top and bottom faces of each model, placed in an three dimensional array of slices, stacks and vectors (i.e. [slices][stacks][3]) were firstly looped over to form a top face using GL_POLYGON, as demonstrated in the following pseudocode:

```
for each slice

{
     glBegin(GL_POLYGON);

     for each stack of the slice

     {

          glVertex3fv([slice][stack]);

     }

     glEnd();

}
```

This produced a rendered face for the top of every model. To render the remainder of the sides, a similar looped technique was employed, demonstrated as follows:

```
glBegin(GL_QUAD_STRIP);

for each stack

{

     glVertex3fv([0][stack]);

     glVertex3fv([1][stack]);

}

glVertex3fv([0][0]);

glVertex3fv([1][0]);
```

```
glEnd();
```

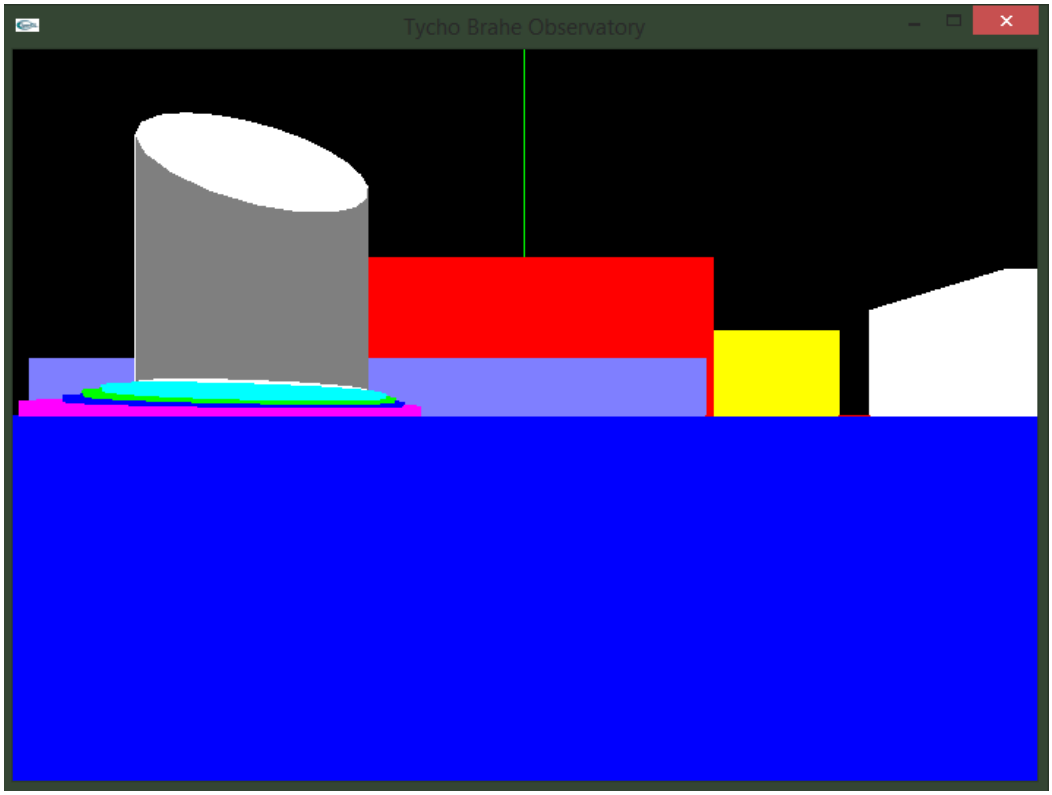


Figure 2.3:
Initial rendering of the scene.

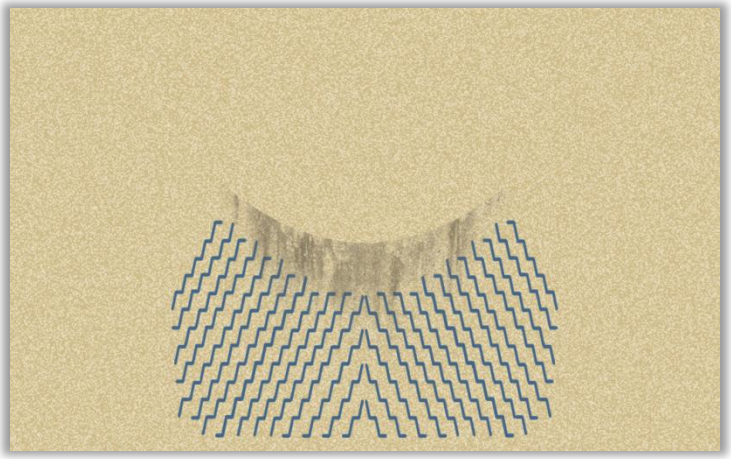
Faces were then culled and depth calculation performed through the enabling of GL_DEPTH_TEST and GL_CULL_FACE.

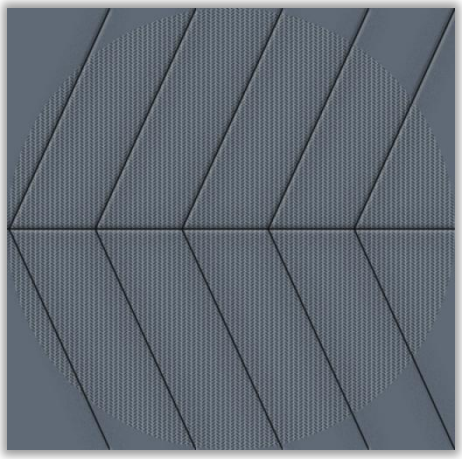
Texture Patterns:

Texturing was initially completed using the SOIL library with images derived directly from the example photograph supplied in the assignment specification; using Photoshop and Microsoft Paint, samples of the image were crafted into textures that were mirrored vertically and horizontally and implemented into the program.

A later revision implemented the custom bitmap parser library, and two alternate cylinder textures were composed using Adobe Photoshop.

Figure 2.4: Revised Cylinder Ring (left) and Cap (right) Textures





A large cube map texture is mapped onto a large inverted cube, which resembles an infinitely distant sky. The **skybox** object is not translated with the camera, and is only transformed by view rotations. The texture pattern used for the skybox was distributed under a Creative Commons Attribution Non-Commercial Share-Alike license, and was created by Roel Boel at the following URL: http://reije081.home.xs4all.nl/skyboxes/.

Texture Mapping:

Initial Texture Mapping:

Texturing was initially completed using SOIL with images derived directly from the example photograph supplied in the assignment specification; using Photoshop and Microsoft Paint, samples of the image were crafted into textures that were mirrored vertically and horizontally and implemented into the program. Texturing was then completed in a similar fashion as before:

```
glBegin(GL_QUADS);

for each stack

{

     glTexCoord2f(1.0, 1.0); glVertex3fv([0][stack]);

     glTexCoord2f(1.0, 0.0); glVertex3fv([1][stack]);

     glTexCoord2f(0.0, 0.0); glVertex3fv([1][stack + 1]);

     glTexCoord2f(0.0, 1.0); glVertex3fv([0][stack + 1]);

}

glTexCoord2f(1.0, 1.0); glVertex3fv([0][stacks]);

glTexCoord2f(1.0, 0.0); glVertex3fv([1][stacks]);
```

```
glTexCoord2f(0.0, 0.0); glVertex3fv([1][0]);
glTexCoord2f(0.0, 1.0); glVertex3fv([0][0]);
glEnd();
```

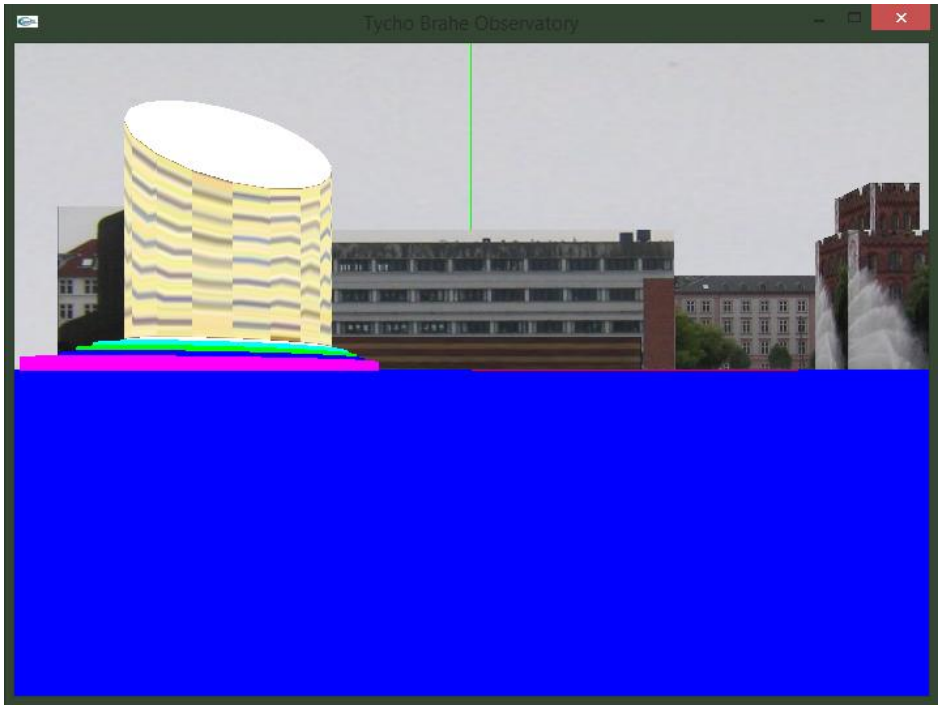


Figure 2.6:
Initial texturing of the scene.

Revised Texture Mapping:

Texture mapping on the cylinder's joining facets was revised to eliminate the texture distortion caused by the linear stretching of the pattern along the Y axis. The U component of the texture coordinate is derived simply from the interpolated angle, θ, in unit form. The coordinate's V component is uniformly zero across the lower non-slanting ring, and is proportional to the height subtracted by the slope on the sloping ring. The relationship can be expressed as:

Lower Side: $u = \frac{\theta}{2\pi}, v = 0.0$

Top Side: $u = \frac{\theta}{2\pi}, v = 1.0 - \frac{h + s + O_y - y}{h + s + O_y}$

$0 \leq \theta \leq 2\pi$

Where h is the height of the solid, s is the slope amplitude, O_y is the origin's Y component, and y is the height of the vertex after computing the slope contribution.

Texture mapping for the cylinder's two caps is performed simply by obtaining the polar coordinates of the current vertex on the XZ plane as a positive unit vector. Both caps have uniform texture coordinates.

$$u = \frac{1}{2} + \frac{1}{2}\cos\theta$$

$$v = \frac{1}{2} + \frac{1}{2}\sin\theta$$

Lighting:

The scene is composed of one directional light source (sun), and a red spot light source below the cylindrical building. Normal vectors orthonormal to the cylinder's joining ring surface were averaged by taking the sum of the surrounding three normals and dividing by three; the resulting quotient is the mean of the three normals.

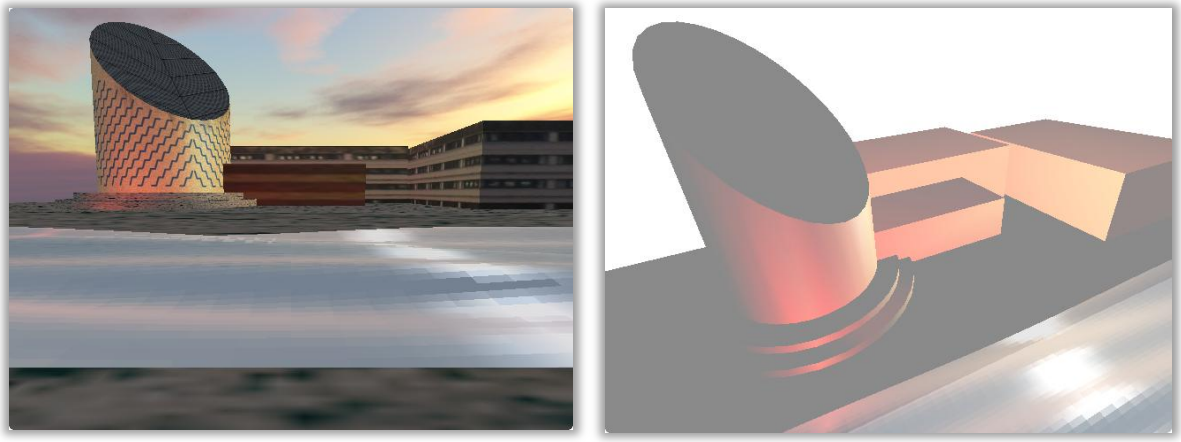
The cylindrical building is given a highly specular surface for technical demonstration purposes. The cylinder specularly reflects the directional light, as well as the spot light whose direction is colinear to the cylinder.

Keyboard Controls

The view rotation may be manipulated using the mouse while holding the left mouse button, and the camera's orbit may be adjusted with the keyboard controls: 'X', 'Y', and 'Z' which orbit on the X, Y and Z axes respectively.

Screenshots:

Figure 2.7: Final Render With (left), and Without (right) Texture Bindings:



Part C: Wavefront OBJ Model Loader

Introduction

Part C demonstrates a custom Wavefront OBJ model parser library, as well as several advanced OpenGL extensions including GLSL Shaders (v3.30 Core Profile) and Vertex Array Objects. Part C utilises the OpenGL Extension Wrangler library to interface with, and obtain function addresses of, OpenGL 3.x extensions.

The OBJ model loader is contained in a separate C++ header file <*objLoader.h*>.

The program will load any arbitrary Wavefront OBJ model that supplies vertex, normal and texture coordinates. To load a custom OBJ model that fits the criteria, or to load one of the models supplied in the project's *Models* directory, drag the model onto the program's icon.

Lighting

The fragment shader program computes the ambient, diffuse and specular contributions of two Phong directional lights, as well as a '**rim light**' which is the inverse dot product of the Normal (*N*) and View (*V*) vectors, raised to a power (R_{power}). The Rim Contribution ($R_{contrib}$) is computed as:

$$R_{contrib} = (\, hermite(\, 1.0 - V \cdot N \,) \,)^{R_{power}} \times R_{albedo}$$

The rim light contribution is interpolated across a Hermite function using GLSL's *smoothstep()* function, which creates a smooth falloff.

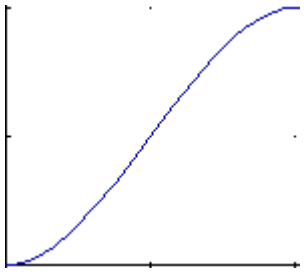


Figure 3.1:
Hermite Curve Sketch
http://cubic.org/docs/hermite.htm

Directional light diffuse and specular contributions are inferred using standard Phong normal vector interpolation and $R \cdot V$ dot product computation. The Blinn-Phong optimization has not been implemented. The viewing vector (*V*) is obtained by taking the negated view-space coordinate of the vertex.

Light diffuse contributions ($D_{contrib}$) are computed as the sum of all diffuse contributions, where *nLights* is the night count, *N* is the surface normal vector, L_i is the current light, $L_{i\ albedo}$ is the current light's diffuse albedo, and *texture* is the diffuse texture sample at the current fragment.

$$D_{contrib} = \sum_{i=0}^{nLights} N \cdot L_i \times L_{i\ albedo} \times texture$$

The program was initially created with only one diffuse light, and a rim contribution. Reviewal of the specification initiated a revision of the program to compute two lighting contributions with unique specular and diffuse contributions.

Viewing Volume

A short viewing volume was created with a Field of View of 60^o, a near plane of 0.01 units, and a far clipping plane which extends into 10.0 on the Z axis. It is anticipated that the program will display small OBJ models.

The viewing frustum is constructed using a custom implementation of the perspective projection matrix. The matrix is expressed as follows, where *N* is the near clipping plane, *F* is the far clipping plane, *fov* is the Field of View expressed in radians, and *aspect* is the aspect ratio of the viewport:

$$\begin{bmatrix} \frac{\cot(0.5\ fov)}{aspect} & 0 & 0 & 0 \\ 0 & \cot(0.5\ fov) & 0 & 0 \\ 0 & 0 & \frac{F+N}{F-N} & 0 \\ 0 & 0 & -1 & \frac{2FN}{F-N} \end{bmatrix}$$

The custom projection matrix is recomputed when the view is resized, and is manually multiplied by each vertex coordinate within a Vertex Shader.

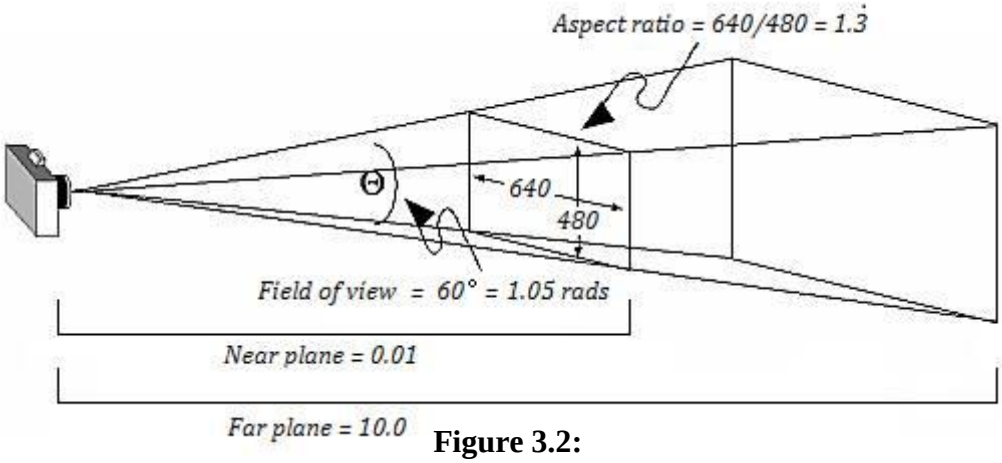


Figure 3.2:
Part C Viewing Volume

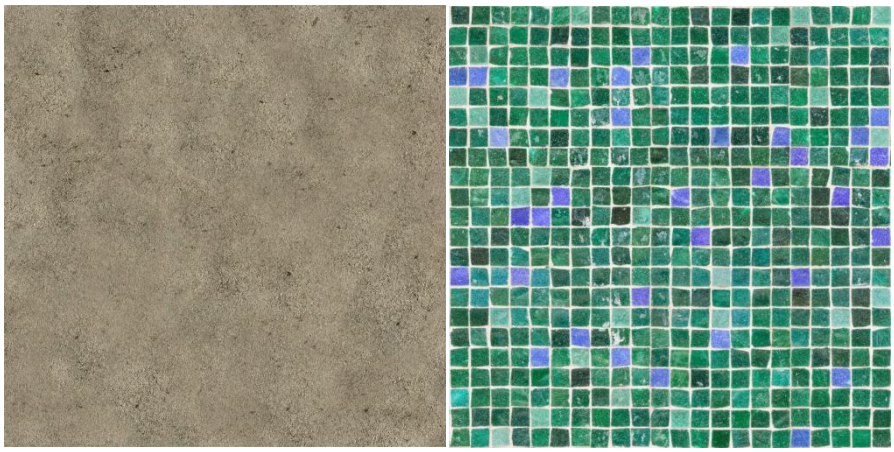
Data Structure

Object vertices are stored in a Standard Template Library vector of eight-dimensional floating point arrays, of which three elements are a vertex coordinate, two are a two-dimensional texture coordinate, and three are a normal vector. The vector stores vertices in an un-indexed form at the expense of memory, so as to allow for efficient Vertex Array Object buffering for the three vertex components.

Texture Patterns

Two texture patterns are provided in the project's *Textures* directory, of which one texture is bound to the object at a time. It is possible to swap texture bindings using the right-click menu.

Figure 3.3: Two Provided Texture Patterns

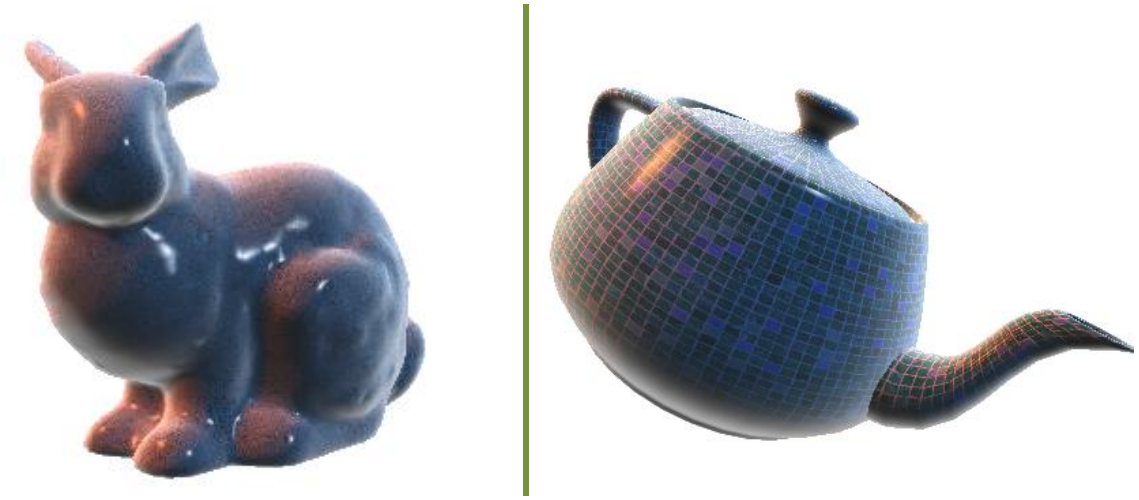


The included texture patterns were provided courtesy of http://www.cgtextures.com/, copyright Marcel Vijfwinkel & Wojtek Starak.

Screen Shots

Figure 3.3: Single-Light Revision

Figure 3.4: Multiple Light Source Revision



Reflection:

Yiannis Chambers

Achievements:

Through the course of this assignment, our achievements in the assignment were varied and many, each challenging but satisfying to accomplish. In summary, the accomplishments of our project that we are most proud of include:

- The successful implementation of Task 3.
- The initial individually design and completion of Task 1 before the release of the course content related to it

Difficulties:

Despite our success, our project was not without difficulties, both technical and conceptual, which were eventually surpassed to give the final product submitted.

Summarised, these difficulties were:

Conflicting Designs

Our work on the assignment began before Week 6, in which the correct methods of boundary representation and geodesic construction were covered in the lecture for that week. By that point, initial designs and constructions of the icosohedron and cylindrical tower had already been completed; the icosohedron had been crafted by both team members using the Golden Rectangle and Offset Pentagon Algorithms respectively, and work had already begun on texturing the same.

The difficulty, then, came with the introduction to correct and efficient construction techniques in the Week 6 lecture; this required a complete modification of existing work and construction procedures, resulting in the abandonment of the Golden Rectangle prototype

entirely - the Offset Pentagon prototype was more easily adapted to correct procedures, which resulted in the development and implementation of the final Geodesic Algorithm.

Thus, due to early progress, work had to be redone due to initial designs conflicting with correct solutions.

Limitations due to Course Structure

Another similar issue arose due to tasks in the assignment requiring a full knowledge of the subject content before attempting some, or all, of the tasks. Team members were unable to work on the keyboard input or menu system, or texture mapping, until Week X, in which this content was first revealed to the class. Thus, progress in the assignment was limited to the release of lecture notes and laboratory relevant to that week's content.

On another note, it was not specified in the Assignment Specification that the knowledge required to complete Task 3 was not to be covered in the course; this was only corrected through individual research by a team member into its implementation and completion. This was disconcerting initially, but once the hurdle was passed, concern over this aspect subsided.

Time Management

Time management was not a difficulty due to ineffective scheduling, poor teamwork or major stumbling blocks – our group started the assignment work early, and consistently added to the project each week.

The main problem was mainly in the amount of work to organise and complete. The hefty workload (90 to 120 hours in total) needed to complete this assignment was staggering to compete with the demands of other equally content-heavy subjects. If our team had not begun work on the assignment as early as we had, we would not have been able to submit the assignment on the correct due date. Even then, work and tweaking of the project continued until the final week of the assignment.

Lessons:

In any development process, new insight into development is garnered through the unique experiences that have shaped the development process. This assignment was no different; the following major lessons were gleaned from working through the various tasks:

Teamwork is invaluable:

This assignment stressed the importance of effective teamwork to a fine degree – without cooperation and delegation of the overall workload, the assignment may not have been completed in its entirety due to its hefty requirements. Working in a team format, where each member is assigned a specific task to complete, and tasks may be worked on cooperatively to compare differing perspectives on the most effective implementation of a set task, is the best method in which to tackle large workloads.

Effective communication is essential:

On similar lines to the previous point, effective teamwork would not be possible without efficient communication between team members, and easy access to cooperative work. Our team stayed in constant communication due to the usage of Facebook and Google Drive as intermediary platforms on which to share and discuss work. This made the exchange of draft code and documents and updates on progress efficient and easy, to the advantage of the development process.

Deinyon Davies

Achievements

Work on parts A and C of the assignment commenced early in the assignment time frame, and as such, several algorithms for coordinate computation and the like were developed before their discussion in lecture material. A prototype Icosahedron generation and drawing algorithm was developed before re-implementing the program with a clearer understanding of the technical requirements.

Part C of the assignment implements OpenGL extensions including GLSL shaders for advanced lighting and rendering implementation, and Vertex Array Objects for efficient and modern data storage and buffering. The implementation demonstrates a comprehensive understanding of transformations (through the manual transformation matrix construction and multiplication), lighting (through a custom implementation of Phong Lighting, Phong Rasterization, and Rim Shading), and texture sampling.

All components of the major assignment implement a custom-written Bitmap image parser library for texture loading. The custom implementation reduces the overhead of a multiple file-format library implementation, and demonstrates an advanced comprehension of texture and texel formats in computer science and OpenGL.

Part B of the assignment implements a unique water simulation algorithm with a time-dependent modulation, specular reflection and blending.

The assignment has been implemented modularly, allowing for simple modification and debugging. The custom three-dimensional geometric vector implementation allows for simple computation of vertex coordinates and vectors.

Difficulties

Some difficulties were experienced in implementing varying light sources in Part B - light positioning became arbitrary and difficult to debug. Helper functions were used to prototype light positions, directions, and albedos.

Normal interpolation between Part B's wave facets was made difficult as a result of the existing implementation, and could not be updated.

Lessons

Though many components of the assignment were developed early in the assignment timeline, and most over the two-week break period, time management proved to be a pressing issue nonetheless. Effective team work and communication lead to a successful final implementation with little stress and frustration.

Numerous object representation and storage techniques were learnt throughout the development of the assignment, as well as geodesic representation algorithms. The assignment was challenging, engaging, exciting, and rewarding to complete.

References:

Tycho Brahe Planetarium - Copenhagen 2005, photographed by Alphalphi, Wikipedia, viewed April 20th 2014, <http://commons.wikimedia.org/wiki/File:TychoBrahePlanetarium-Copenhagen.jpg>

R. Boel, 2006, Sun altitude of 5 degrees above the horizon, Roel z'n Boel, viewed April 19th, 2014, <http://reije081.home.xs4all.nl/skyboxes/>.

M. Vijfwinkel, Starak, W, CGTextures, viewed April 14th, 2014, <http://www.cgtextures.com/>.