# Temple on Mars

Zoe Wall
40182161@live.napier.ac.uk
Edinburgh Napier University
Computer Graphics (SET08116)

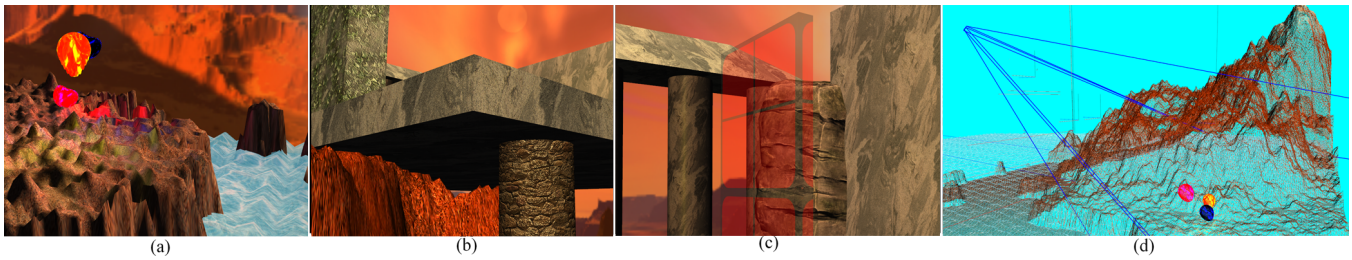


**Figure 1:** *Project screenshots. (a) Point lights with Vertex Displacement and Parent-Child Hierarchy, (b) Normal Mapping, (c) Transparency and Render Ordering, and (e) View Frustum Culling and Wire-framing.*

## Abstract

This project aims to implement a 3D scene, rendered in real-time using OpenGL and C++. The project was intended to be aesthetically interesting, but more importantly intended to show an understanding of fundamental computer graphics principles. This report details the industry techniques used within the development of the outdoor Mars scene.

**Keywords:** skybox, hierarchy, lighting, OpenGL, GLSL

## 1 Introduction

**Figure 2:** ***Project Inspiration - Destiny Mars Landscape*** *- [Bungie 2014]*

**Project Aims** The main aims of this project were not to produce something photo-realistic but to show understanding of the key concepts of computer graphics. The setting of the scene is a form of landscape on Mars with the inclusion of some extra-terrestrial objects and strange effects. The initial ideas were that of a barren land filled with broken stone and large metallic structures, littered with weeds and with dust clouds. The main inspiration for this type of scene comes from the vast landscapes used within Destiny (see Figure 2).

**High Level of Detail** Using different texturing techniques, a high level of detail is hope to be achieved. For example normal mapping to give the impression of more shape and detail to an object. Along with blend mapping and alpha-mapping used on different objects in the scene.

**Figure 3:** ***Normal Mapping example - Fallout 4*** *- [Bethesda Game Studios 2015]*

Even with the improvements to GPU hardware in recent times, there is always a need for optimisation. In this scene optimisation techniques such as a scene graph for spacial partitioning was used alongside view culling.

## 2 Implementation

### 2.1 Parent-Child Hierarchy

A scene graph in the form of a parent-child hierarchy was implemented within the scene to partition the scene's space into smaller parts to allow for optimization and easier control over the overall development of the scene. The scene graph contains all of the geometry for the particular scene, it makes transformations and rendering easier as a parent-child hierarchy is such that every object in the scene inherits from a single root. This means that each child object's own model matrix is relative to it's parent's (see Listing 1).

**Listing 1:** *Parent-child Hierarchy Update Method*

```
1  if (parent){
2     if (parent->myType != sky && myType != sky)
3     {
4        mworld = parent->mworld * mlocal;
5     }
6  }
7
8  for (auto &e : children)    // iterate through list of children
9  {
10    Obj* child = e.second;
11    child->normalMatrix = normalMatrix;  // set normal matrix ↩
       for child object
12    child->update(this, delta_time);     // recurse
13 }
```

This technique is not only useful for positioning the geometry into world space, it is also useful for rendering and culling. For example, a test can be made to see if an object's parent is not visible within the view frustrum, then it should assume that the child is also not visible and continue without performing the calculations. Rendering and updating is done by traversing the n-tree that makes up the hierarchy. Firstly the root is rendered if visible. It then iterates through a list of it's children rendering each in turn. If a child of the root is invisible, it will not check to see whether the children are, and the whole branch will be culled.

## 2.2 Lighting

Lighting in computer graphics is very important to get correct as it is the main basis of making a scene look realistic. Lighting calculations are performed on the GPU by programs called shaders. There are three programmable shaders within the graphics pipeline: the vertex, the geometry and the fragment shader. There are two main methods for lighting a scene using OpenGL: Gouraud shading, and Phong shading. Gouraud shading is calculations performed per geometry vertex. This means that an object in a scene is coloured by interpolating the colour value between each vertex, as opposed to the more expensive but better Phong shading. Where colour calculations are performed on the fragment shader on a per pixel basis. (See Figure 4)

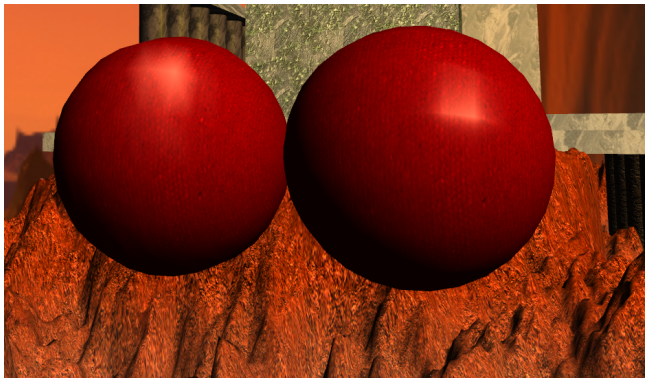

**Figure 4:** ***Implementation of both Gouraud and Phong shading within the project.*** *- The sphere on the left is Gouraud shaded, note the specular reflection is less concentrated and accurate, whilst the sphere on the right is Phong shaded.*

The main lighting types used within the scene:

- Ambient Constant lighting which is representative of light from a fixed-intensity and fixed-coloured source. It is used to raise the overall brightness level of every object within the scene.
- Diffuse Directional lighting, percentage brightness according to the angle of the light and vertex.
- Specular Reflective lighting, what makes an object appear to be shiny or reflective.
- Attenuation Light from a source that degrades over distance such as a point or a spot light.

The equation for ambient lighting is as follows:

$$DA \tag{1}$$

where $D$ is the diffuse reflection of the material and $A$ is the ambient intensity of the scene. The diffuse reflection is the coloured reflection component of the material.

Diffuse lighting is a little more complex as the intensity of the light is not constant. It relies on the direction of the light source to the normal of the geometry. In simple Gouraud shading, the transformed normal of each vertex is calculated, and the angle between that and the light direction is calculated using the scalar product. If the vertex is facing the light source, the intensity of the light will be 100% and however, 90 degrees away from the light, the intensity drops to zero.

$$DAmax(LN, 0) \tag{2}$$

Specular lighting is dependent on the object's material. It is the bright highlight that is dependent on the direction of the eye, in this case the position of the camera. How reflective the material is depends on how bright and how large this highlight appears on the object. In computer graphics it is a very important element in creating a realistic lighting model, as it mimics the reflective properties of a surface. For this calculation, the position of the eye (or camera) must also be taken into account.

$$SCmax(HN, 0)^m \tag{3}$$

where:

$$V = \frac{eyePosition - worldPosition}{||eyePosition - worldPosition||} \tag{4}$$

Where the eye Position is the position of the camera, and the world position is the position of the vertex in world space, and H is the half vector, V plus the direction of the light normalised.

Attenuation lighting is where the lighting is taken from a point source or a spot source, where the intensity is multiplied by a factor of its range. In the real world lighting model, light from a point source drops off exponentially, however attenuation values are set for artistic purposes, to give a more even fall off.

## 2.3 Texturing

### 2.3.1 Vertex Displacement

Vertex Displacement Mapping or simply Displacement Mapping is a technique allowing to deform a polygonal mesh using a texture (displacement map) in order to add surface detail. The scene includes a procedurally generated terrain from a height map

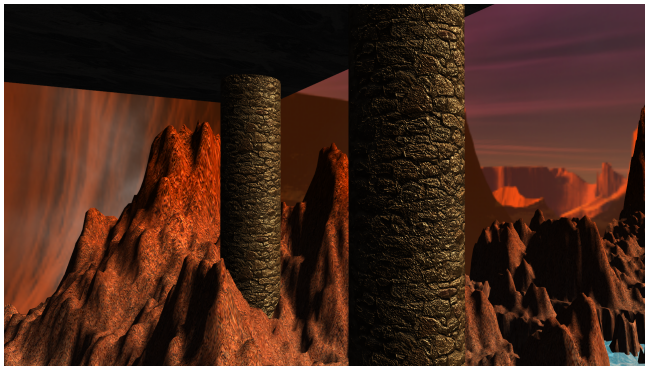

**Figure 5:** ***Normal Mapping Example from Scene*** *- These pillars are completely smooth cylinder objects, however note how by using a normal map, another level of detail is added making it appear that the pillars are made from stone, with cracks and bumps.*

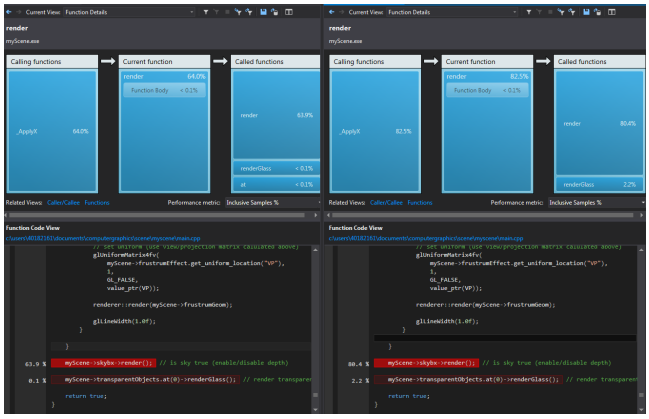

**Figure 6:** ***Comparison of Profiling Results*** *- The results on the left are after optimization, it shows the hot path render() to be roughly 20% lower than the original results.*

### 2.4 Normal Mapping

#### 2.4.1 Skybox

A skybox is essentially a set of textures called a cubemap. These are seamless textures that fit together like a layout of a cube A skybox is a cube rendered from the inside out, with the depth buffer turned off. It also needs to be transformed with the camera, so it gives the effect that it is infinitely large and far away.

### 2.5 Transparency

To render a transparent object, OpenGL has a mechanism called blending, which combines any colour already in the frame buffer with the colour of the primitive, and the resulting blended colour is stored back in the framebuffer [Jerome Guinot 2001]. This causes problems for rendering a scene in real-time as render order must be taken into account. This is because if the transparent object is rendered first, the texture behind the object is blended to it, which means that any object rendered after the glass will be occluded by the object. To solve this issue, an enumeration was used to check for object type when rendering. If the object rendered is a glass object it is rendered separately after the main render function.

## 3 Optimization

#### 3.0.1 Profiling

Originally, with no optimisation techniques implemented, running the performance profiler within Visual Studio 2013 showed that the most expensive call path based on the sample counts was the render() function called to traverse down the hierarchy and render each object in turn.

The rendering function initially used 83.20% of inclusive samples, meaning it was the most expensive function overall. After applying the optimisation techniques, the profiler returned 63.9% for the main render function meaning the frustum culling saved on average 20% of the program's performance. This can also be noted when monitoring the frame-rate for the scene, with the frame-rate clipped, the frame-rate stays at a constant 60FPS. Features such as exploring the environment, and toggling wire-frames on and off make no impact on this frame-rate whatsoever. When running without a clip, the frame rate can be seen to improve when looking away from a complex part of the scene which shows that the view clipping is working, as there is less operations per frame and therefore an improved frame-rate. The slowest parts of the code apart from the render function are part of the geometry builder in the set-up of the scene. There is a peak at the start of the program of CPU usage. Further analysis shows that the the hot paths here are both the function to generate the terrain, and the function generating the plane for the water mesh. In both cases, the functions are expensive due to the high number of polygons required to make the mesh look realistic.

### 3.1 Geometry Shader

The geometry shader is the second part of the graphics pipeline. Unlike the vertex or fragment shaders, the geometry shader is used to transform the actual geometry of an object. A primitive is passed through the shader, for example a point, and the geometry shader then can either add or remove vertices to change the overall shape of the mesh. In the scene, the geometry shader was used to draw the radius of the bounding spheres for each object in the hierarchy. Each mesh has a selection of positions on the vertex array buffer, to calculate the radius a function iterates through positions to find the largest point of the geometry, the length of this as a float is passed into the geometry shader along with the centre point, and a line is drawn.

For the particle effect the geometry shader is used in conjunction with transform feedback. A transform feedback is a technique used to capture geometry data before it is rendered. The primitives passed to the geometry shader are first transformed and saved to the transform feedback buffer. This buffer is then used in the second render pass, to output the particles to the screen. Within this project, the second render pass also utilises the geometry shader using a technique called billboarding.

Billboarding is a technique which allows a texture to be rendered always in the direction facing the user. This was used within the second render pass of the particles to render a smoke texture to each, seen in Figure 7.

### 3.2 View Frustum Culling

View frustum culling is an important optimisation technique which essentially means to remove objects that are out of the view of the camera. By calculating the frustum planes of the camera, intersection tests can be performed to see if an object's bounding volume is visible. Any object that is classified as outside of the view frustum

**Figure 7:** ***Screen Capture of Smoke Effect*** *- This effect uses a transform feedback buffer and a billboarding technique to render the texture facing the user.*

is culled - not rendered.

A geometric approach was taken to extract the planes from the view frustum, this involved determining the eight points defining the corners of the near and far plane, and use these points to define all six planes. [Lighthouse3d 2015] The cross product of these corners are then used to find the normal pointing inwards for each plane. Once the planes have been extracted, the plane equation can be used to determine whether the object is inside or outside of the plane. (See equation 5)

$$D = \hat{n} \cdot (x_0 - x_i) \tag{5}$$

Where $D$ is the signed distance from a point $x_0$ to a plane that contains the point $x_i$. $\hat{n}$ is the normalised plane normal. For the intersection test, this equation is repeated for each plane, if the distance is negative for any plane then the centre point lies outside the frustum. To account for objects that intersect with the plane, the test approximates the point to be within the frustum if the distance is less than the radius of the bounding sphere of the object, see Listing 2

**Listing 2:** *Intersection Test for View Frustum*

```
for (int i = 0; i < myScene->planeNormals->length(); ++i)
{
  // for each plane check if  intersection occurs
  vec3 pointOnPlane;

  if (i < 3)
  {
    pointOnPlane = myScene->planePoints[ftl];      // first three planes are far, top and left therefore corner is in all three
  }
  else
  {
    pointOnPlane = myScene->planePoints[nbr];
  }

  vec3 centre = vec3(getWorldPos());

  float d;
  d = dot(myScene->planeNormals[i], centre - pointOnPlane);

  if (d < - getRadius())
  {
    //cout << "CULLING! " << this->myName << endl;
    visible = false;
    break;
  }

}
```

In the scene the intersection tests are performed using bounding spheres. The use of bounding spheres as volumes is limited due to the fact most objects within the scene are not spheres, for example the columns will have a very wide radius to account for the height, however they will still be classified as within the frustum if horizontal and within it's length, even though it is not visible. However this is a reasonable enough approximation to use as it is less expensive to compute and still noticeably optimizes the render function.

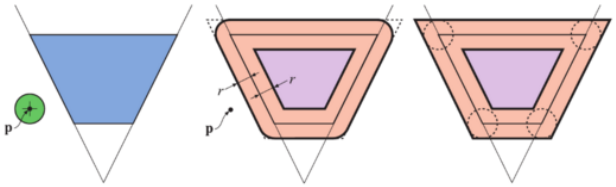


**Figure 8:** ***Diagram of Approximation of View Frustum*** *- The figure shows a bounding sphere and a frustum. The diagram on the right shows the approximation used in the calculation. If a sphere intersects within the radius boundary of the frustum plane it will be classified as inside the plane. It can lead to incorrect classifications of objects outside the rounded corner, however it is reasonable enough for this case. [Akenine-Moller et al. 2008]*

## 4 Conclusion

In conclusion, the first attempt at building up the basics of the Mars scene was successful. It showed many core principles of computer graphics, in particular optimisation techniques with view frustum culling, and texturing effects for more than just basic decoration. Future work will be to further the lighting and optimisation techniques to recreate light more realistically. For example to show how light bends as it passes through a transparent object, and use Screen Space Ambient Occlusion to add further level of detail.

## References


AKENINE-MOLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering*, 3 ed. AK Peters, Ltd.

BETHESDA GAME STUDIOS. 2015. Destiny.

BUNGIE. 2014. Destiny.

JEROME GUINOT. 2001. Vertex displacement mapping using glsl. *Accessed: March 2015.* www.opengl.org.

LIGHTHOUSE3D. 2015. Tutorials: View frustum culling. *Accessed: March 2015.* www.lighthouse3d.com.