Coursework Report

Paulius Bieksa
40216180@napier.ac.uk
Edinburgh Napier University - Computer Graphics (SET08116)

1 Abstract

This project aims to implement a three dimensional scene graphics scene, using c++ and GLSL. While one of the aims was to make the scene aesthetically interesting, the main focus is on understanding the main concepts and techniques of graphics programming and the rendering pipeline.

Keywords – OpenGl, GLSL, Phong, Graphics, Portal, Stencil buffer, Colour correction

2 Introduction

Project Aims The aim of this project is to show the understanding of the graphics pipeline, shading and simple shading models. The scene is set in a small alley in a dreamlike setting. While there are no particular examples of inspiration to point out, the portal effect was largely inspired by the game portal.

Effects The main effect used in this project was a multi-pass rendering using the stencil buffer to create portals. A computationally cheap lighting solution was used, namely Blinn-Phong lighting (figure 1) with addition of simple shadows and normal mapping. Transform hierarchy was used to better manage multi-object contraptions in the scene. Materials were used to create a more realistic look of copper by adjusting the spectrum of reflected light. Additionally a colour correction post-processing effect was implemented.

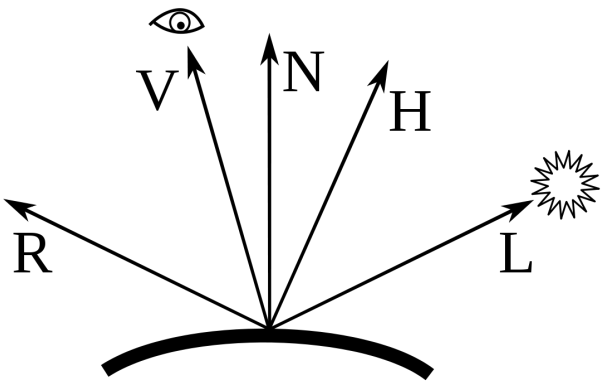

Figure 1: **Blinn-Phong reflection model**

3 Implementation

Object hierarchy Handling objects comprised of multiple meshes is a tedious process. To remedy this, this project uses object hierarchy. The hierarchy is achieved by creating a class that inherits the functionality of a pre-existing mesh class in the framework provided and adds a parent pointer and some helper functions.

Blinn-Phong shading One of the simplest and most computationally inexpensive methods of simulating lighting in a virtual environment is the Blinn-Phong model. This lighting technique has three main components: ambient diffuse and specular, all three of which are calculated using camera position, light position, and the surface normal.

Shadows The shaders in this project is also able to calculate simplistic shadows. While at this stage of the project it can only calculate shadow for only one type of lights, it demonstrates the technique adequately. The shadow effect is achieved by first rendering the scene to a depth buffer using the source of light as the camera. A projection of the buffer onto the scene is then used to find which parts are lit.

Material shading Materials and normal mapping are used in this project to augment the Blinn-Phong shader to create a more photo-realistic look. Materials change how objects reflect different lighting components, while normal mapping adjusts the object normals based on a texture that uses RGB values to represent the adjusted normal vector. These adjusted normals are then sued to calculate the colour of each fragment.

Listing 1: Normal mapping in the fragment shader

```
1   vec3 norm_sample = texture(normal_map, tex_coord).xyz;
2   new_normal = normalize(normal);
3   vec3 new_tangent = normalize(tangent);
4   vec3 new_binormal = normalize(binormal);
5
6   // Transform components to range [0, 1]
7   norm_sample = 2.0 * norm_sample - vec3(1.0, 1.0, 1.0);
8   // Generate TBN matrix
9   mat3 TBN = mat3(new_tangent, new_binormal, ↩
     new_normal);
10  // Return sampled normal transformed by TBN
11  new_normal = normalize(TBN * norm_sample);
```

3.1 Portal rendering

Stencil buffer The approach to render portals chosen for this project was to use the stencil buffer to mark out areas of the screen and render to those ares selectively. Unlike the depth buffer, the stencil buffer consists of user defined integers. Rendering can be set to only occur in areas where specific values are written in the stencil buffer. After enable the editing of the stencil mask the marking of the screen region is done using normal openGL draw commands. In this case the meshes for the portals were rendered to mark screen space areas to later render the scene to (figure 2 and 3).

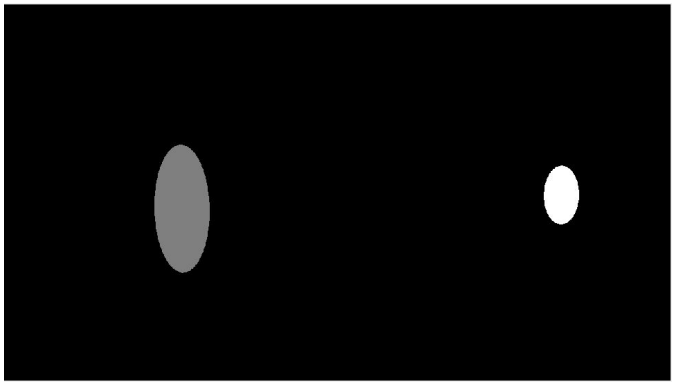

Figure 2: **Stencil buffer**

Figure 3: **Image rendered using the stencil buffer**

Frame buffer object While the framework provided has a frame buffer object, it does not provide stencil buffer attachment support. Therefore a different solution had to be used to implement any multi-pass rendering techniques and keep the functionality of the frame-buffers. Instead of using the frame buffer class provided this project implemented a frame buffer object using openGL. A depth-stencil attachment was used due to only stencil attachment being hazarded against in the openGL documentation.

Rendering image seen trough portal The image seen trough the portals is what would be seen when looking out of the other portal (figure 4). To achieve this, when rendering, all objects in the scene were transformed with an offset matrix in addition to the normal transformation by the MVP matrix. The order of operations here is very important. The offset matrix itself was comprised of the transform matrix of the 'input' portal (figure 4, marked in orange) and the inverse of the transform matrix of the 'output' portal (figure 4, marked in blue) multiplied in that order. This way the rotation of the portals is applied first and translation afterwards. Another problem is not rendering objects in front of the 'output' portal after the transformation. With only the offset transformation applied, depending on the position of the camera some object that should not be shown can be rendered (figure 4, marked in green). To solve this problem an additional calculation in the shader had to be done to discard all texels that are preceding the portal plane. This is done by simply using the dot product to check the angle between the portal normal and the texel location from the portal as origin (listing 2). Note that the offset used in the shader is actually the inverse in calculating the positions of the objects seen trough the portals.

Listing 2: Calculating which texels to discard

```
1  if (dot(eye_pos − portal_pos, portal_normal) < 0)
2  {
3     if (dot(other_portal_normal, ((offset ∗ vec4(portal_pos, 1.0)↩
       ).xyz − position)) > 0)
4        discard;
5  }
6  else
7  {
8     if (dot(other_portal_normal ∗ −1, ((offset ∗ vec4(↩
       portal_pos, 1.0)).xyz − position)) > 0)
9        discard;
10 }
```

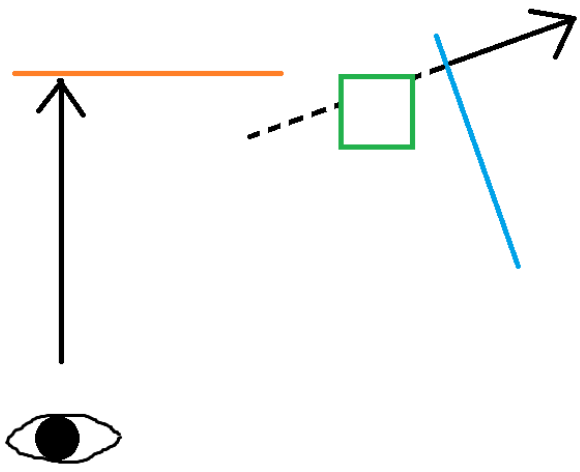

Figure 4: **Top-down view of portals**

3.2 Post-processing

Colour correction This project implements colour correction which lets the user adjust hue, saturation and brightness values. To achieve this the rendered image is captured in a buffer as a render pass, and used as a texture. In the shader the colour of each pixel is transformed into a cylindrical colour coordinate system where hue is represented by the angle in degrees starting at red, saturation by the distance from the center and brightness by the height in the cylindical colour space (figure 5). The system used in this project is a modified HSV (stands for hue, saturation, value) model. While the HSV model represents brightness (V component) as the highest of the RGB components, this project instead uses a more visually relevant luma value. Luma is the weighted average of gamma-corrected R, G, and B, based on their contribution to perceived luminance.

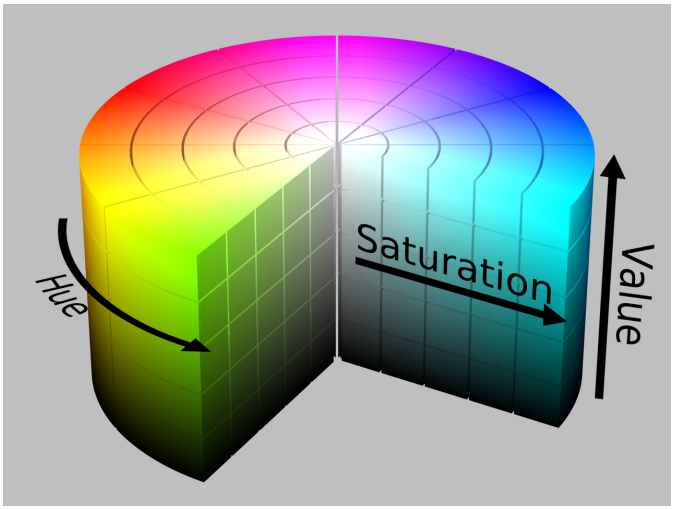

Figure 5: **HSV colour model**

Masking Masking is a simple effect to add certain textures to parts of the screen. This project uses it to show the user a controls 'cheat-sheet'. Images with relevant information are mixed with the render based on the alpha value of the masking images to create an overlay.

3.3 Optimisation

Having the most openGL calls the render function is unavoidably the most intensive. However, with some optimisation this project runs smoothly even on an integrated graphics card. The first optimisation technique used was just removing all unnecessary openGL calls. A lot of these calls can be done once per rendering of the scene rather than once per object rendered. Some openGl calls, like setting the texture for the skybox, can be done once, because openGL saves all uniform information with the shader. Another technique used was discused earlier in this paper, where unwanted pixels are discarded instead of calculating lighting and texturing information. While optimisation was not that shader's primary purpose it does save some gpu time which translates in to a smoother running program.

4 Future Work

Due to time constraints and the scope of this project some techniques were not fully fleshed out or even attempted. In the future the shadowing can be expanded to be cast by all light sources. The shadow map for the directional light would have to use orthogonal projection. Lighting and shadowing trough portals can be done with some tweaking of the maths, generating an occluder mesh and passing some additional information to the shaders. Particles can be added swirling around the portals to improve the esthetic.

5 Conclusion

In conclusion, the goals of this project have been achieved. This coursework has provided an opportunity to learn a lot of the skills required to implement the graphical techniques taught in the course.

Figure 6: **Normal renderl**

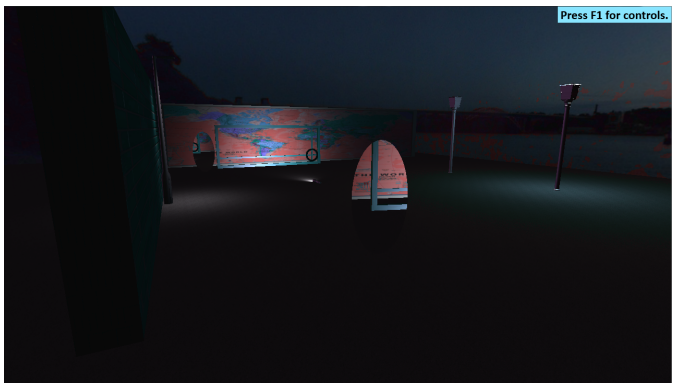

Figure 7: **Hue shifted**

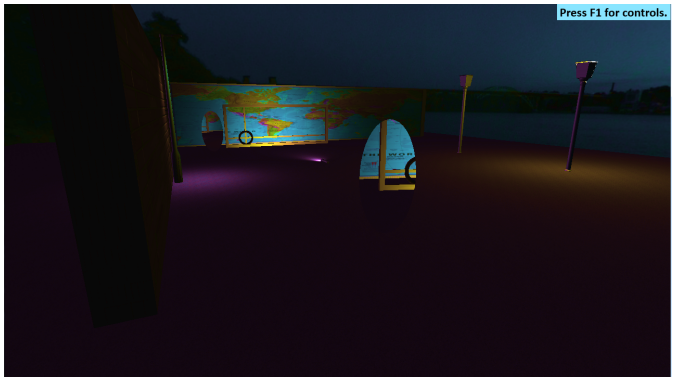

Figure 8: **Render with saturation turned up**

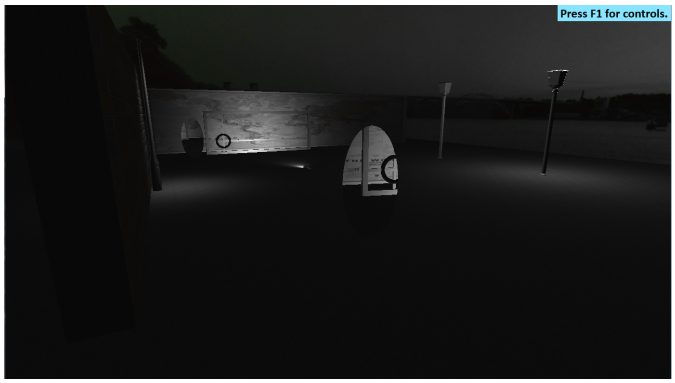

Figure 9: **Render with saturation turned to minimum**