Computer Graphics Coursework Report

Marcos Ramirez Aceves
40284094@live.napier.ac.uk
Edinburgh Napier University - Computer Graphics (SET08116)

Abstract

The aim of this project is to create a 3D scene rendered in real time using the OpenGL framework provided. Related to the technical part, the project is inspired by previous years projects that can be found in Edinburgh Napier University Games Development website, while the visual aspect of the scene is inspired by Artorias tombstone which is a location of the video game Dark Souls. Different techniques have been implemented in the development of this project, starting from the most basic elements of Computer Graphics seen in the lectures and the workbook, to further implementation outside that scope. This report covers the techniques used and their implementation to create the scene, future work, optimization and conclusion.

Figure 1: Artorias Grave

Figure 2: Artorias Grave

Keywords – Computer Graphics, Rendering, Shadows, Instancing, Texturing, Graphics, Lighting, Shading

1 Introduction

The main goal of the project is to demonstrate the understanding of the basic concepts of Computer Graphics and the capacity of going beyond that scope and provide further implementation. The 3D scene is formed by the following elements:

- Island terrain surrounded by lava.
- A spot light and a point light.
- A "futuristic" object to demonstrate transforms hierarchy.
- A tomb, columns and a sword in a rock.
- Instancing swords around the sword in the rock.
- A skybox.

The Computer Graphics techniques used in the elements above are the following:

- Texturing using normal mapping.
- Texturing using blend mapping.
- Instancing.
- Shadow mapping.
- Percentage Close Filtering for shadows(PCF).
- Noise.
- Masking and thermal vision post-processing.
- Terrain generation using height maps.
- Lighting.
- Transforms hierarchy.
- Multiple cameras.

There are three types of cameras in the scene, a free camera that allows the user to move freely around the scene, a target camera to see the scene from different positions and an arc ball camera that allows the user to move around the "futuristic" object to see in detail the transforms hierarchy.

The following sections will describe in more detail the techniques mentioned above and how they have been implemented in the scene.

2 Related Work

The basic techniques used in this project are based on the Computer Graphics Workbook, even though most of them have been partially modified in order to fit the purpose of the scene, the basic idea is still the same.
About the further implementation, for the thermal vision, masking was mixed with a post-processing effect to create something more complex while for instancing and PCF(Percentage Close Filtering) online tutorials and research was needed.

3 Implementation

3.1 Lighting

Two different types of lights are being used in the scene, a spot light and a point light. A point light has a position in space and emits light in all directions up to some range/distance, point lights are useful for simulating lamps and other local sources of light.

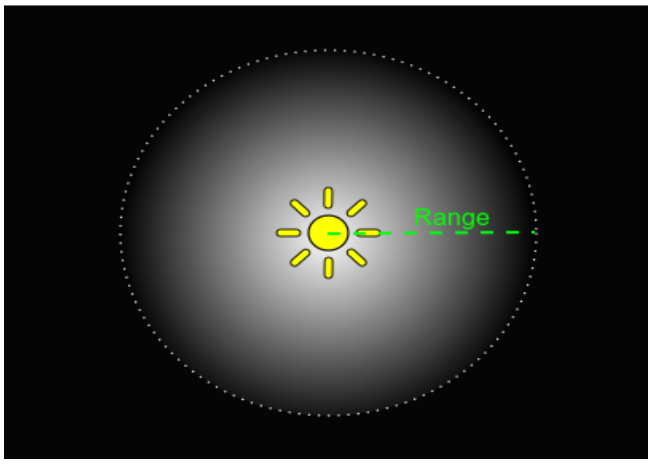


Figure 3: Point Light

A spot light also has a location and range, but in addition it has an angle and direction. Spots lights are used for artificial light sources like flashlights or searchlights.

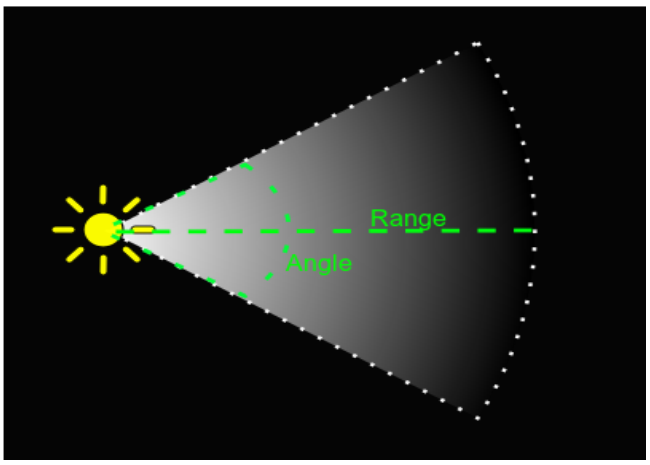


Figure 4: Spot Light

The illumination model used for the different objects in the scene is the Blinn-Phong model.

The Phong model takes into account the ambient, diffuse and specular reflection of light. Also it supposes an improvement in comparison with the Gouraud model providing better results. In this model, the surface normal is interpolated between each vertex and the colour is calculated for each normal. However, the Phong model has some issues which are that it does not accommodate the reflection equation, ambient illumination is a hack and it is physical incorrect (does not conservate energy).

$$I = I_a K_d + f_a t t I_s (K_d (N \cdot L) + K_s (R \cdot V)^n)$$

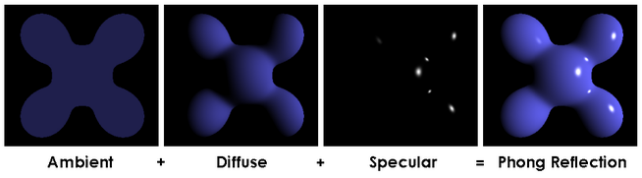


Figure 5: Phong Components

To solve the problems mentioned above we need the Blinn-Phong illumination model that uses the "half-way" vector.

$$I = f_a t t I_s K_s (N \cdot H)^n$$

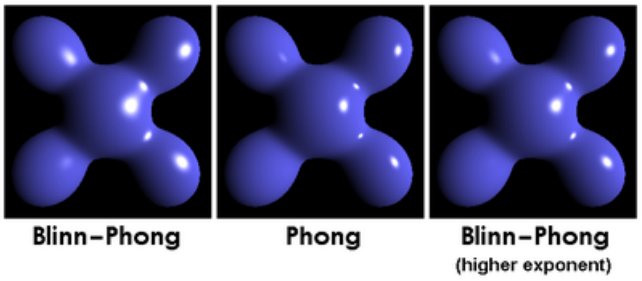


Figure 6: Illumination models comparison

Listing 1: Lighting Code

```
1   //Calculate diffuse component
2   float k_diffuse = max(dot(surface_normal,light_direction)↩
    ,0.0f);
3   vec4 diffuse = k_diffuse * (light_colour * ↩
    material_diffuse_reflection);
4
5   //Calculate specular component
6   vec3 half_vector = normalize(light_direction + ↩
    view_direction);
7   float k_specular = pow(max(dot(surface_normal,half_vector↩
    ),0.0f),material_shininess);
8   vec4 specular = k_specular * (light_colour * ↩
    material_specular_reflection);
9
10  //Calculate final colour
11  vec4 primary = material_emissive + diffuse;
12  vec4 colour = (primary * texture_colour) + specular;
13  colour.a = 1.0f;
14
```

3.2 Texturing

There are two approaches for texturing, the first one is from data using texture mapping and the second one is procedural using shaders. In this project two texture mapping techniques are used, blend mapping and normal mapping.
Texture mapping follows a number of stages which are the projector function stage, the corresponder function stage, the obtain value stage and the transform value function stage. The general principle of texture mapping is to unwrap a 3D object on a 2D texture.

Blend mapping basically consist in detirmining the colour of a pixel by blending/mixing two textures together using a blend map. This texture mapping is used for the columns in the scene.

Listing 2: Blend Mapping

```
1   // Sample the two main textures
2   vec4 colour1 = texture(texture[0],texture_coordinates);
3   vec4 colour2 = texture(texture[1],texture_coordinates);
4
5   // Sample the blend texture
6   vec4 blend_map = texture(blend,texture_coordinates);
7
8   //Mix the main samples using r component from blend value
9   vec4 tex_colour = mix(colour1, colour2, blend_map.r);
10  colour = tex_colour;
11
```

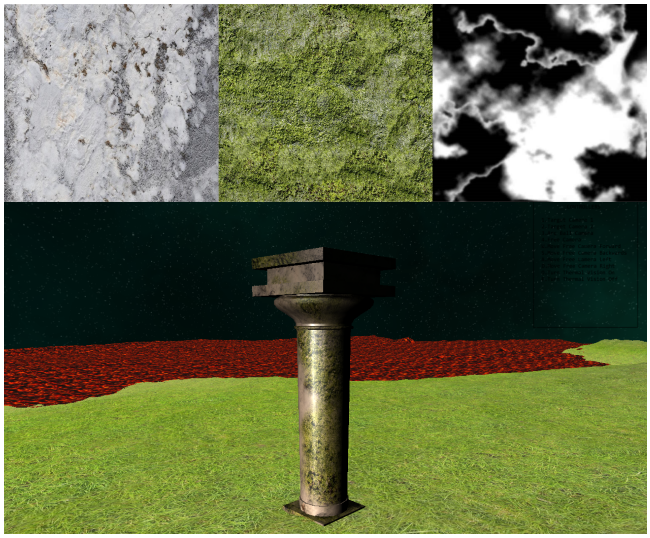


Figure 7: Blend Mapping

Normal mapping calculates the surface normal on a pixel level using a normal map. This allows us to provide a good level of detail in the objects with out adding more polygons, basically with this technique the objects surface will not look flat. The RGB values in the normal map represent the(x,y,z) components of the normal at that point in the texture. And in addition to the surface normal, we need the tangent and bi-normal vectors. Normal mapping is used in the grave, the swords, the rock and the "futuristic" object.

Listing 3: Normal Mapping

```
1   // Ensure normal, tangent and binormal are unit length (normalized)
2   normal = normalize(normal);
3   tangent = normalize(tangent);
4   binormal = normalize(binormal);
5
6   // Sample normal from normal map
7   vec3 sampled_normal = texture(normal_map,texture_coordinate).xyz;
8
9   // Transform components to range [0, 1]
10  sampled_normal = 2.0 * sampled_normal - vec3(1.0, 1.0, 1.0);
11
12  // Generate TBN matrix
13  mat3 TBN = mat3(tangent, binormal, normal);
14
15  // Return sampled normal transformed by TBN
16  return normalize(TBN * sampled_normal);
17
```

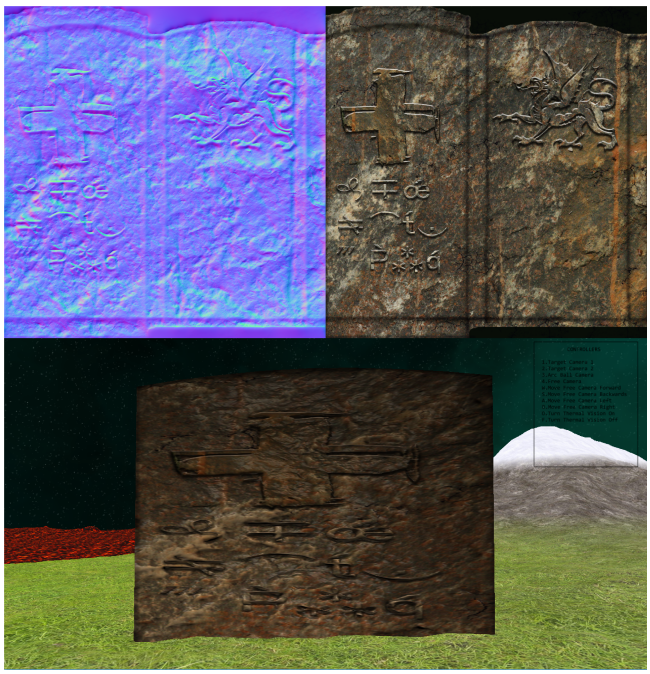


Figure 8: Normal Mapping

3.3 Transforms Hierarchy

Transforms hierarchy is a simple approach to make objects move along with other objects (into other things). It basically consist in multiplying the model matrix of the object being rendered with the model matrices of the previous objects. Doing this we can create a chain of transformations (translation, rotation and scaling), making connections between objects in a parent-child hierarchy. This technique is implemented in the "futuristic" object.

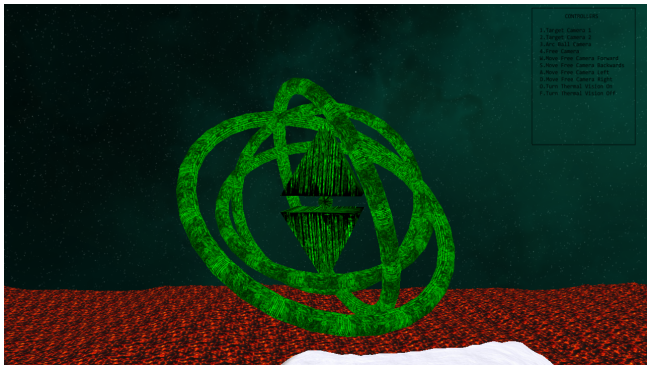


Figure 9: Transforms Hierarchy

Listing 4: Transforms Hierarchy

```
1       for (size_t i = 0; i < pyramid_meshes.size(); i++)
2       {
3           auto M = pyramid_meshes[i].get_transform().↩
        get_transform_matrix();
4           //Apply hierarchy chain
5           for (size_t j = i; j > 0; j--)
6           {
7               M = pyramid_meshes[j - 1].get_transform().↩
        get_transform_matrix() * M;
8           }
9           //Create MVP
10          auto MVP = PV * M;
11      }
12
```

3.4 Terrain Generation

Terrain generation consist in converting the colour of the pixels from a texture to height data on a plane (the pixel colour represents the y component of the vertex), the whiter a zone is the higher it will be. Basically we read the texture data to generate vertex position, allowing us to generate terrain as geometry. The texture we used for that is called heightmap, which is just a texture with height data. Terrain generation is being used for the island and the lava in the 3D scene, in addition, the lava uses a UV SCROLL to simulate that is moving.

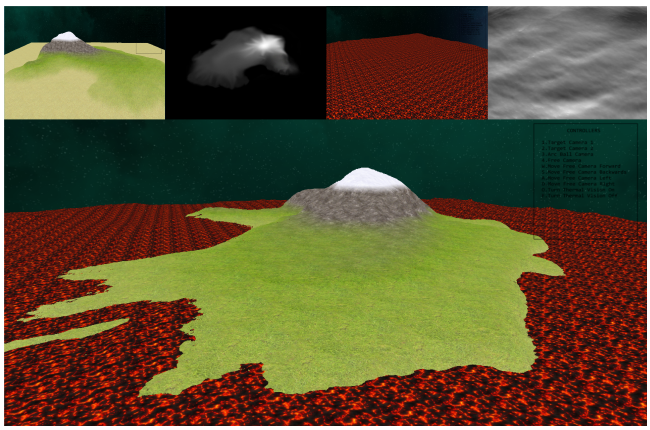


Figure 10: Terrain

3.5 Shadowing

There are three approaches for shadowing: projection shadows, shadow maps and volume shadows. For reasons of complexity and time the shadowing approach taking in this project is shadow mapping.

In shadow mapping casting shadows is seen as a visibility problem, we need to compare the distance of the rendered points in the scene to the light source with the values stored in the shadow map. There are two stages in this technique. The first one is the shadow map generation, basically what we need to do is rendered the scene from the light source's point of view, performing the rendering to the depth buffer and storing the rendered scene in a texture called shadow map. The shadow map contains information about the distance of the illuminated surface points to the light source.

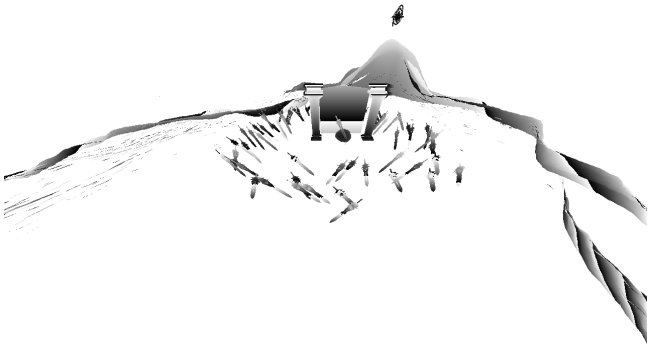


Figure 11: Shadow Map

The second stage consist in rendering the scene from the camera's point of view and every time a pixel is being rendered we need to compare the position of that pixel with the value stored in the shadow map to check if it is in shadow or not.

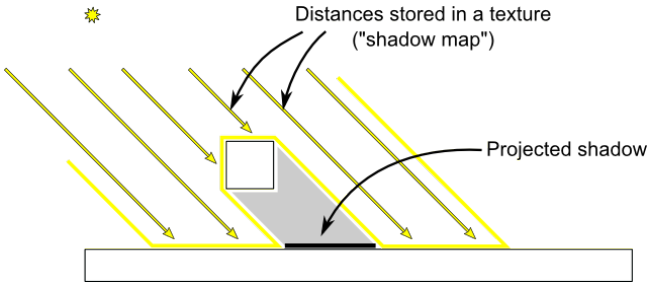


Figure 12: Object In Shadow

One of the problems of shadow mapping is the field of view of the shadow map, as an object to shadow may be outside it, however in this project a spot light is used for shadows therefore this was not an issue.
Apart from that, some of the problems encounter when performing this shadowing approach were shadow acne, "peter panning" and aliasing. Shadow acne causes the objects to self shadow in an incorrect way while peter panning causes to make shadows look like "disconnected" from the objects.

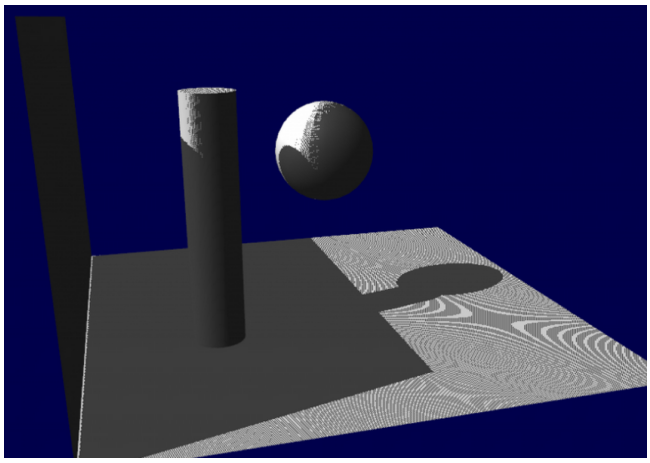


Figure 13: Shadow Acne and Peter Panning

This two problems are easily solved by adding a bias (in order to make it more efficient the bias depends on the light direction). Even if this is supposed to be easy, selecting the correct bias can be a bit difficult.

Listing 5: Bias Calculation

```
1       //Bias calculation
2       vec3 light_direction = normalize(light_position - proj_coords);
3       float bias = max(0.000005* (1.0 - dot(normal, light_direction)), 0.0000095);
4
```

The main problem with this shadowing approach is aliasing, which causes the shadows to be rendered with jagged edges and therefore they look really bad, the reason why that happens is that the resolution of the shadow map is too low and solving it can be a complex and difficult task. Several approaches to solve this issue were taking into account like VSM (Variance Shadow Mapping), ESM (Exponential Shadow Mapping) or PCF (Percentage Close Filtering), the approach taken for this project is PCF.

3.5.1 Percentage Close Filtering

The basic idea of PCF is that the visibility (shadow factor) of a pixel is calculated by taking into account the visibility of a number of surrounded pixels and working out the average. PCF is an easy way to make shadows to look softer. In this project 64 samples of surrounding pixels are being use to calculate the shadow factor, for the simple reason of providing better results.

0.48	0.48	0.48	0.49	0.5	0.5	0.51
0.48	0.48	0.48	0.49	0.5	0.5	0.51
0.48	0.48	0.49	0.49	0.5	0.51	0.51
0.49	0.49	0.49	0.5	0.51	0.51	0.51
0.49	0.49	0.5	0.51	0.51	0.51	0.51
0.49	0.5	0.5	0.51	0.51	0.51	0.51
0.49	0.5	0.51	0.51	0.51	0.51	0.51

Figure 14: PCF

Listing 6: Shadow Factor Calculation

```
1       //Initial visibility
2       float visibility = 1.0;
3
4       //Modify visibility acording to the surrounded pixels
5       for (int i=0;i<64;i++)
6       {
7           int index = int(64.0*random(position*1000.0,i))%64;
8           if ( texture(shadow_map, shadow_tex_coords + poissonDisk[index]/700.0 ).x  <  z + bias )
9           {
10              visibility-=0.0135;
11          }
12      }
13      return visibility;
14
```

This approach provides better results than the initial one taken from the Computer Graphics Workbook, however the jagged edges are still noticeable. To improve that, a random function is used to select the sample and as a result a noise effect is produced in the shadows making the jagged edges even less noticeable and providing softer shadows.

Listing 7: Random Function For Noise

```
1       float random(vec3 position, int i)
2       {
3           vec4 seed4 = vec4(position, i);
4           float dot_product = dot(seed4, vec4(12.9898,78.233,45.164,94.673));
5           return fract(sin(dot_product) * 43758.5453);
6       }
7
```

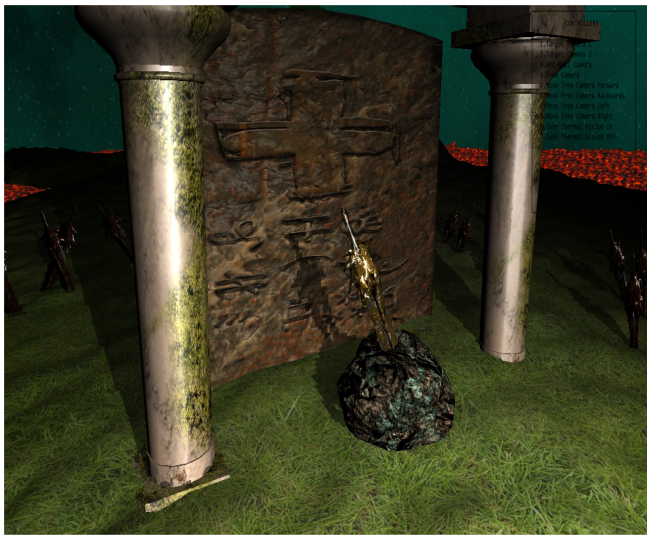


Figure 15: Shadows With Out PCF and Noise

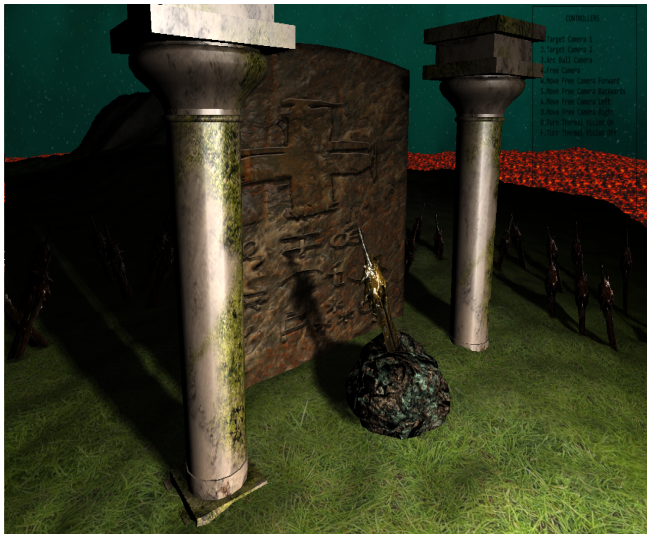


Figure 16: Shadows With PCF and Noise

As can be seen in the figures above, in the first one the shadows are implemented with out PCF and noise while in the second one PCF and noise are implemented and the improvement in the shadows is easily noticeable.

3.6 Instancing

When rendering a 3D scene if we want to render several instances of the same model the simplest way of action is to create a loop and perform a render call for each instance. The main problem that we have with this is what if we want to render thousands of instances? In that case the CPU will reach a performance bottleneck, even if the GPU can render those instances pretty fast, giving the commands from the CPU to the GPU every frame is very costly and not so fast.

Instancing allows us to render multiple instances of the same model with a single drawing call and therefore solving the issue mentioned above. One of the problems that we need to think about when performing instancing is the position of each instance, as now we are going to render all the instances with a single render call we need a way of providing to each instance at least a different model matrix so they are rendered in different positions, we can also set other attributes to be different but in this project we are only interested in providing a different position for each instance. We can do this by setting a uniform that contains an array of matrices and use the glInstanceId() method to select the relevant matrix for each instance, however this is rather inefficient. An alternative is to store all those model matrices in the model VAO.

The VAO contains information of each vertex in a model in attributes list, in fact we have been using this all the time but in an indirect way as that functionality is provided by the framework. Those attributes are accessed in the shaders by using "in" variables, and each time a vertex is rendered the set of variables change. In the case of instance rendering, that process is going to be repeated for each instance.

In the VAO we can store two types of attributes, per vertex attributes and per instance attributes, the framework provide to us 5 per vertex attributes. For this project we only need one per instance attribute, the model matrix, so the instances have different positions when they are rendered. The problem is that the largest size possible of data to be included on the VAO attributes list is of 4 float and therefore we need to split the model matrix in 4 attributes, one for each of the matrix columns.

Listing 8: VAO

```
1  //Bind VAO
2  glBindVertexArray(model_meshes["blade"].get_geometry().↩
   get_array_object());
3  //Generate VBO for instancing
4  glGenBuffers(1, &position_vbo);
5  glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
6  //Enable VAO attributes to store the model matrix ↩
   instances
7  glEnableVertexAttribArray(5);
8  glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, sizeof(↩
   glm::mat4), (void*)0);
9  glEnableVertexAttribArray(6);
10 glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(↩
   glm::mat4), (void*)(sizeof(glm::vec4)));
11 glEnableVertexAttribArray(7);
12 glVertexAttribPointer(7, 4, GL_FLOAT, GL_FALSE, sizeof(↩
   glm::mat4), (void*)(2 * sizeof(glm::vec4)));
13 glEnableVertexAttribArray(8);
14 glVertexAttribPointer(8, 4, GL_FLOAT, GL_FALSE, sizeof(↩
   glm::mat4), (void*)(3 * sizeof(glm::vec4)));
15 //Attributes change per instance
16 glVertexAttribDivisor(5, 1);
17 glVertexAttribDivisor(6, 1);
18 glVertexAttribDivisor(7, 1);
19 glVertexAttribDivisor(8, 1);
20 glBindVertexArray(0);
21
```

As shown above, we need to bind the VAO of the mesh that we are going to create instances from, and then enable the VAO attributes to store the model matrix columns, also we need to specify that those attributes change per instance and not per vertex.

Once we have the VAO set we need to generate the data (the model matrices), that later on when performing the rendering (after setting the uniforms), we will load to a VBO, which is what we use to store the data in the VAO attributes list. Finally we just need to call the glDrawElementsInstanced() method that OpenGL provides to us for instance rendering and in which we specify the number of instances to be rendered.

Listing 9: Instance Rendering

```
1  //Bind VAO
2  glBindVertexArray(model_meshes["blade"].get_geometry().↩
   get_array_object());
3
4  //Bind VBO
5  glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
6
7  //Load VBO data
8  glBufferData(GL_ARRAY_BUFFER, model_matrices.size() ↩
   * sizeof(glm::mat4), &model_matrices[0], ↩
   GL_STATIC_DRAW);
9
10 //Render instances
11 glDrawElementsInstanced(GL_TRIANGLES, model_meshes[↩
   "blade"].get_geometry().get_index_count(), ↩
   GL_UNSIGNED_INT, 0, model_matrices.size());
12
```

Instance rendering is used to render multiple swords around the one stuck in the rock. All those swords have the same material properties and textures but they are loaded with different model matrices so they are rendered in different positions as can be seen in the figure below.

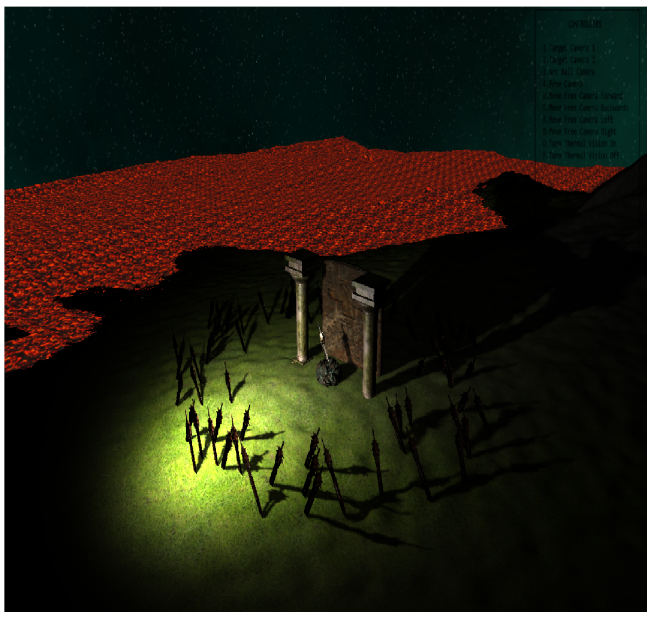
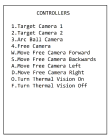


Figure 17: Instancing

3.7 Post-processing

Post-processing is a technique that allows us to capture a render in a texture and then used it in the future, the captured texture is called render pass. In a bit of more detail, the process is similar to shadow mapping, for this project the first thing we need to do is render the scene to a frame buffer where the captured texture is going to be stored, and later we are going to use that texture to render a quad that covers the complete screen and apply the selected effects.

Two post-processing effects are used in this project, the first one is a simple "user interface" that shows the controllers for moving the camera, changing between cameras and turning on and off the thermal vision effect. The implementation of this is fairly easy, as it consist just in a masking post-processing effect. To implement it we just need to mix the texture captured in the frame buffer with an alpha map, which in this case is a simple white image with the controllers in black.



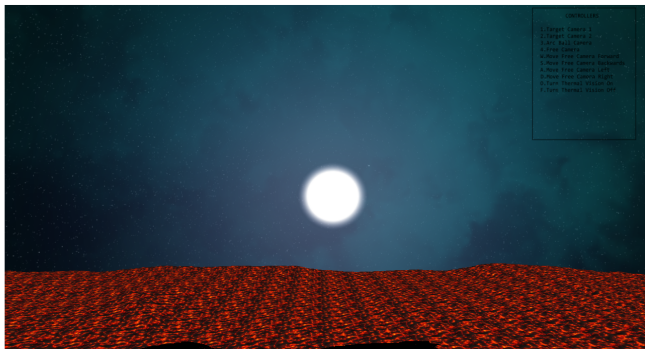


Figure 18: Controllers

The controllers can be seen at the top right of the screen.

Listing 10: Masking/Controllers

```
1   // Sampler used to get texture colour
2   uniform sampler2D tex;
3   // Alpha map
4   uniform sampler2D alpha_map;
5
6   // Incoming texture coordinate
7   layout(location = 0) in vec2 tex_coord;
8   // Outgoing colour
9   layout(location = 0) out vec4 out_colour;
10
11  void main()
12  {
13    // Sample textures
14    vec4 tex_colour1 = texture(tex,tex_coord);
15    vec4 tex_colour2 = texture(alpha_map, tex_coord);
16
17    // Mix textures for colour
18    out_colour = tex_colour1 * tex_colour2;
19
20    //Ensure colour.a is 1
21    out_colour.a = 1;
22  }
23
```

The second post-processing effect is a "thermal vision" effect, to implement it we need to make a gradient which in this case is a blue-yellow-red gradient, make the thermal map texture (substitute pixel luminance for temperature), get pixel's temperature from thermal map texture and map it to gradient value. In addition this is mixed with a masking effect to achieve the final result.

Listing 11: Thermal Vision

```
1   // Sampler used to get texture colour
2   uniform sampler2D tex;
3   // Alpha map
4   uniform sampler2D alpha_map;
5
6   // Incoming texture coordinate
7   layout(location = 0) in vec2 tex_coord;
8   // Outgoing colour
9   layout(location = 0) out vec4 out_colour;
10
11  void main()
12  {
13    //Calculate thermal colour
14    vec3 tc = vec3(1.0, 0.0, 0.0);
15    vec3 pixcol = texture(tex, tex_coord).rgb;
16    vec3 colors[3];
17    colors[0] = vec3(0.,0.,1.);
18    colors[1] = vec3(1.,1.,0.);
19    colors[2] = vec3(1.,0.,0.);
20    float lum = (pixcol.r+pixcol.g+pixcol.b)/3.;
21    int ix = (lum < 0.5)? 0:1;
22    tc = mix(colors[ix],colors[ix+1],(lum-float(ix)*0.5)/0.5);
23    vec4 thermal_colour = vec4(tc, 1.0);
24
25    //Mix thermal colour with alpha map
26    vec4 alpha_map_colour = texture(alpha_map,tex_coord);
27    out_colour = thermal_colour * alpha_map_colour;
28    out_colour.a = 1;
29  }
30
```

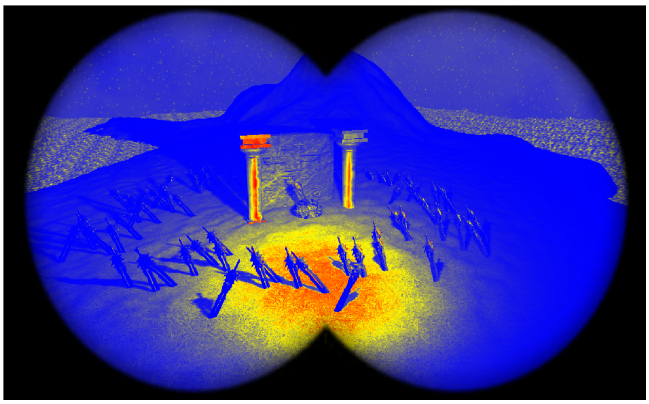


Figure 19: Thermal Vision

4 Optimization

One of the approaches to optimize this project was multiplying the view and projection matrices at the beginning of the render so they could be use for each render call with the corresponding model matrix and in this way reducing the number of matrix multiplications. Also instancing itself is an optimization for rendering multiple objects with a single render call.

Further optimization will require analyzing the code looking for CPU bottlenecks and the parts where it takes more time to be processed and if it is possible improve it.

5 Future Work

For future work, I will mainly focus on making even better shadows. The solution in this project provides better and softer shadows than the original one provided by the Computer Graphics Workbook but they can still be improved. A way would be try to make them to look better when the light source is further away like improving the shadow map resolution or to base the softness proportionally to the distance, another way would be try to implement VSM (Variance Shadow Mapping) or PCSS (Percentage Close Soft Shadows) which are approaches that can provide better results than PCF (Percentage Close Filtering).

Apart from shadows I am interested in particle effects like smoke or fire, and I think they will probably be very interesting elements to implement.

6 Conclusion

During the initial stages of shadowing, I spent over a week trying to figure out the best action plan and what type of shadows should be implemented in relation of complexity-time. Trying to implement VSM (Variance Shadow Mapping) first, I came to the conclusion that it was going to be a bit complex for this project to implement, for that reason the decision was to go with PCF. Even with PCF, several online tutorials proposed similar but also a bit different ways of implementing them, I tried with two PCF approaches and based on the visual results I selected the one that has been finally implemented in this project. With shadows the only problem encountered was the implementation itself as the concepts were pretty clear.

In the case of instancing, there were problems with both the concepts and the implementation. Several researches online were made first to understand exactly how instancing works and when the concepts became clearer I started with the implementation. The main problem with the implementation was the framework itself, as I needed to work with the VAO, VBOs and the rendering calls directly and that functionality was always provided to us via the framework I had trouble in understanding them and making away to adapt that to the framework. However after everything became clearer implementing instancing was pretty simple and straightforward.

In conclusion, the final result of the project is pretty successful as it covers the main elements that were initially planned to be implemented as the advance features, being those elements soft shadows, in this case PCF (Percentage Close Filtering), and instancing, adding to that some post-processing effects. Also, by implementing the features mentioned in the Future Work section, the scene will be more technical interesting and visual engaging.

Link to the video: https://youtu.be/-ui1gyK-V98

References

[1] Tutorial 16: Shadow mapping: http://www.opengl-tutorial.org/es/intermediate-tutorials/tutorial-16-shadow-mapping/

[2] Shadow Mapping: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping

[3] OpenGL 3D Game Tutorial 40: Percentage Closer Filtering: https://www.youtube.com/watch?v=TcbvOvmLSow

[4] Soft Shadows With PCF: http://fabiensanglard.net/shadowmappingPCF/index.php

[5] VSM: http://fabiensanglard.net/shadowmappingVSM/

[6] Instancing: https://learnopengl.com/Advanced-OpenGL/Instancing

[7] Particles/Instancing: http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

[8] OpenGL Instancing Demystified: https://www.gamedev.net/articles/programming/graphics/opengl-instancing-demystified-r3226/

[9] Instanced Rendering: http://www.informit.com/articles/article.aspx?p=2033340&seqNum=5

[10] OpenGL 3D Game Tutorial 36: Instanced Rendering: https://www.youtube.com/watch?v=Rm-By2NJsrc

[11] Instanced Drawing In OpenGL: http://in2gpu.com/2014/09/07/instanced-drawing-opengl/

[12] Thermal Vision: http://www.geeks3d.com/20101123/shader-library-predators-thermal-vision-post-processing-

[13] Edinburgh Napier University Games Development Web Page: http://games.soc.napier.ac.uk/graphics.html