Coursework Report - Part 1
Hero's Rest

Davide Maurilio Morello
40219838@live.napier.ac.uk
Edinburgh Napier University - Computer Graphics (SET08116)

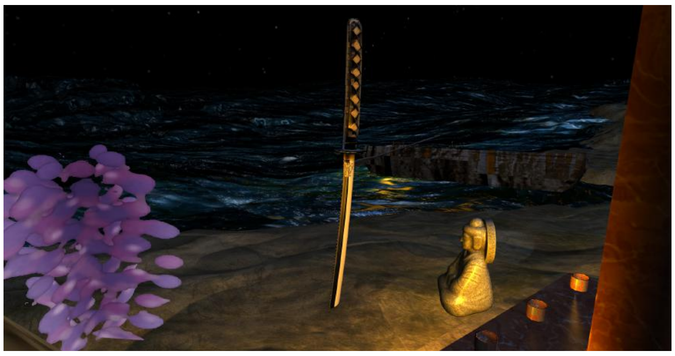

Figure 1: **Katana** - Grave of a Samurai

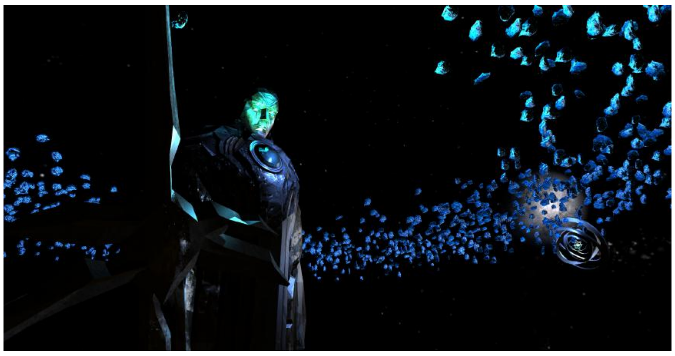

Figure 2: **Guardian** - Towering Guardian Statue

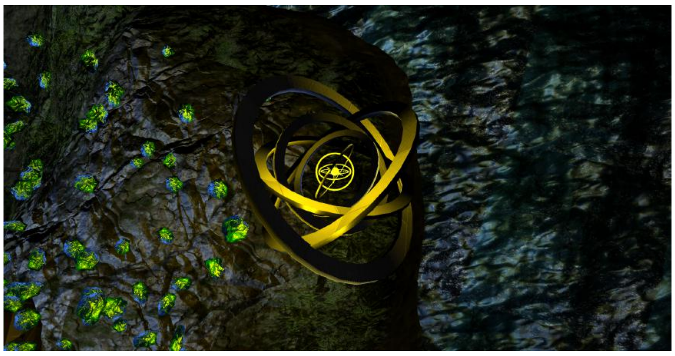

Figure 3: **Amillary** - The rotating amillary, color set to yellow

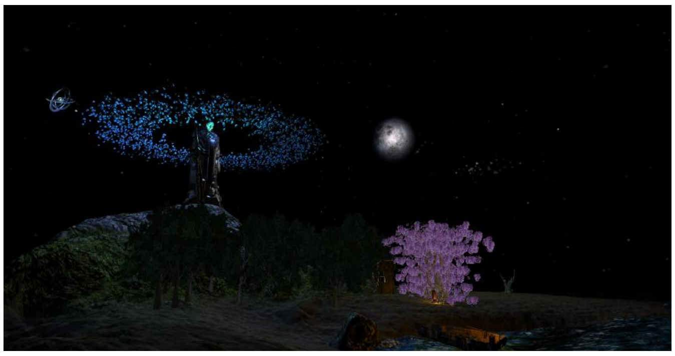

Figure 4: **General View** - The island as a whole seen from the sea

Abstract

The intention and goal of this project was to create an appealing 3D scene capable of rendering in Real Time through the usage of the OpenGL Graphics API supported by a C++ wrapper and the implementation of a custom Graphics Framework. In the design phase of project, many different sources of inspiration. After a lengthy evaluation of pros and cons, the choice landed on the decision of drawn a Fantasy looking scenario inspired by a blend of Lord of the Rings and Japanese culture. This report will cover the fist implementation of the scene, achieved by implementing a large of array of Shaders, lights and camera types and some other basic geometrical and CG techniques.

Keywords – hyerarchical-transforms, shaders, lights, shadows, maps, OpenGL, GLSL

1 Introduction

1.1 Subject and Aim of the project

The main target of the project was to create a realistic scene in order to show understanding and control over some fundamental concepts of Computer Graphics, Real Time rendering and basic understanding of the OpenGL graphics API. The setting chosen is a small island, surrounded by a colossal statue such as the Argonath (see **Figure 5**) which hail to the Fellowship during their sail on the Anduin. Overlooked by statue, a small boat near the shoreline can be found carrying a small lantern shining fiery light on the sea. Similarly, close to the coast the grave of a Samurai and his Katana surrounded by a Violet Tree adoarn the seashore and create a shiny contrast against the other trees that separate the small woods from the the sand. A very bright moon from the top right shines a pale diffused light on a maelstrom of debris and magical Amillary that around the huge guardian statue and its rock looking sword.

Figure 5: **Argonath** -Lord of the Rings

1.2 Development Criteria

The development process of the scene although heavily steered towards a set of the techniques by the main requirements of the project, has tried to keep the goal of delivering a good looking and consistent subject that would feel appropriate as the setting of a small game or least a decent enough setting for a cutscene. For this reason, the overall setting is much larger and than generally required in order to implement the required technologies and many different areas are realistically empty and only procedurally generated, textured and covered. Lastly, a good looking scene requires good looking models. For this reason, most of the scene uses custom meshes to achieve a realistic look rather than represent a abstract environment.

2 Technologies

Overview The project is built modern OpenGL library in conjunction with a C++ wrapper, the OpenGL Mathematics Library (glm) and the usage of GLSL shaders. In order to achieve most of the results, the scene uses the combination of multiple shader files responsible for generating different geometric, color or other effects, this allows the subject to be rendered individually and differently from one another bringing to life a more detailed and realistic scene.

Techniques In order to delivery a basic, yet appealing scene a number of different techniques has been used. The most worthy of note are:

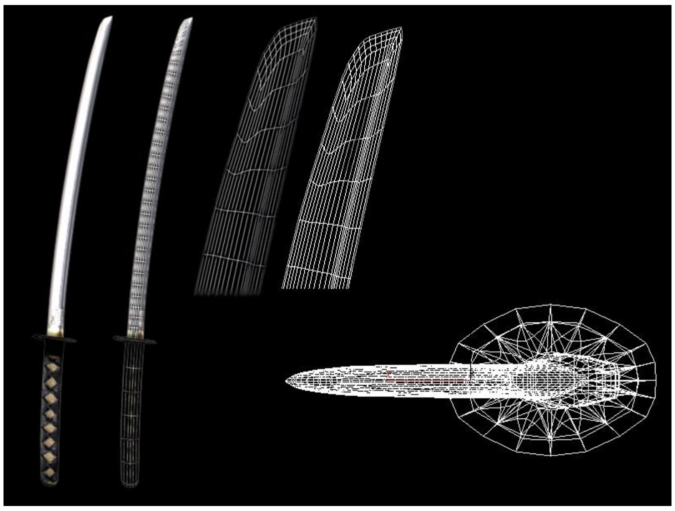
Figure 6: **- One of the various models used to render the scene**

- Lightning

- Parent/Child transform Hierarchy

- Terrain generation through height maps

- Texturing, blending, movement and Normal Mapping

- Multiple Cameras

- Skybox and Cube Maps

- Shadow Maps

And some slightly more advanced techniques that will not be covered in this report.

- Alpha Mapping

- Instancing

This techniques will be briefly discussed singularly, generally many of the subjects in the scene will implement multiple combinations of the above mentioned technologies in order to attain the desired effect.

3 Lightning

The project is developed throughout the implementation of the Blinn-Phong (see **Figure 7**) shading model, a faster and more realistic overhaul of the Phong model modified to attain better precision and computational efficiency. The model, just like its parent relies of three ideal types of light to deliver a realistic effect.

Ambient Light An ambient light in Computer Graphics represents an omni-directional, fixed-intensity and fixed-color light source that affects all objects in the scene equally. This type of light, clearly not existent in nature is used to offer every object in the scene an approximation of the ambient light that should on an object as a result of the reflection coming from neighboring corpses. It is used to raise the overall brightness level of every object and make them more visible.

Diffuse Reflection Diffuse reflection originates from a combination of internal spreading of light on the object, or in other words the light that is absorbed and then re-emitted by the object, and external scattering from the rough surface of the object. This leads to a fairly uniform non direction reflection resulting from the light being reflecting multi-directionally by the hit surface and returning an even matte look to the object.

Specular Highlight A specular reflection or highlight is the result of a light shining into a specularly reflective object shining it back directionally towards the viewer and offering hints regarding the actual space of the surface being touched by the beam and its position in relation it.

The Blinn-Phong is consequently calculated for each light type and the results combined in order to attain a model that follows the Blinn optimization using a half-vector.

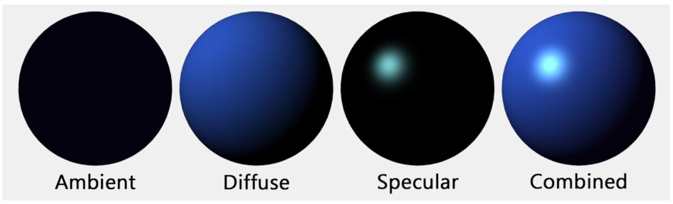

Figure 7: **Phong reflection model** - Ambient, diffuse and specular components contribute to final effect

Listing 1: Blinn Phong Shaders

```
// Calculate ambient component
   vec4 ambient = light.ambient_intensity * mat.diffuse_reflection;

// Calculate diffuse component
 float diffuseK = max(dot(normal, light.light_dir), 0.0);
   vec4 diffuse = diffuseK * (mat.diffuse_reflection * light.light_colour);
   // Calculate half vector
   vec3 half_vector = normalize(light.light_dir + view_dir);

// Calculate specular component
 float k = pow(max(dot(normal, half_vector), 0.0), mat.shininess);
   vec4 specular = k * (light.light_colour * mat.specular_reflection);

// Calculate colour to return
vec4 colour = ((mat.emissive + ambient + diffuse) * tex_colour) + specular;
colour.a = 1.0;
```

In reality however, lights are not necessarily all alike, and in this project we consider three different approximations.

Directional Light A huge, massive light that shines over the scene. Provides ambient light and could be for example identified as the sun in a real life scenario. Due to the scene being dark, this light can be enabled in the project, but the default value for Hero's Rest, is with this main source disabled.

Point Lights Light originates from a single point, and spreads outward in all directions.

This are found in the gravestone candles, boat lantern, moon, rotating amillary, and the guardian statues azure gem.

Spot Lights Light originates from a single point, and spreads outward in a cone. The moon offers the major spotlight of the project, a bright wide code that combines with the azure one originating from the gem of the colossal statue.

Since However the scene offers multiple light sources, such as for example the boat lantern shining a yellow light on the water (see **Figure 8**), which is also hit by the white light of the moon, and the the amillary, in order to achieve the most realistic result the process needs to be repeated of each one of the various sources, simply avoiding the ambient component when the light is a spot or point.

Listing 2: Multiple Lights and Light Types

```
// Calculates direction ambient light
  colour = calculate_direction(light, mat, normal, view_dir, tex_colour);

  // Sum point lights
  for(int i = 0; i < points.length(); i++){
    colour += calculate_point(points[i], mat, position, normal, view_dir, tex_colour);
  }

  // Sum spot lights
  for(int i = 0; i < spots.length(); i++){
    colour += calculate_spot(spots[i], mat, position, normal, view_dir, tex_colour);
  }
```

This structure offers the base building brick of all the shaders used in the project.

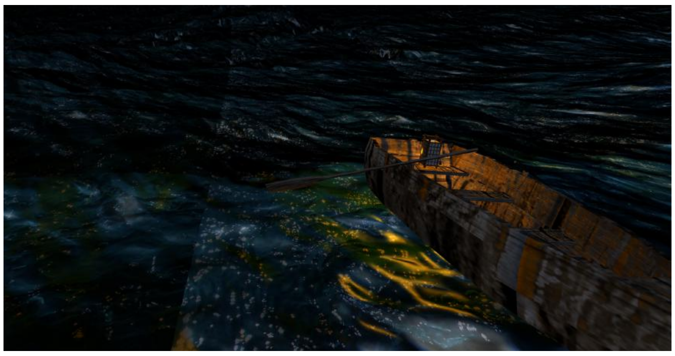

Figure 8: **Phong reflection model** - Ambient, diffuse and specular components contribute to final effect

4 Hierarchy of transforms

In graphics positions, rotation and size of a mesh are captured in a Model matrix originating from the non commutable combination of:

$$[ModelMatrix] = [ObjectTranslation][ObjectRotation][ObjectScale]$$

Due to their mathematical nature, this transformations can be applied hierarchically to objects in a parent-child relation. This link offer the possibility to apply all of the transformations of the parent to the child for a low cost, and with the advantage of seamlessly binding together the movements of the two objects in a rather computationally efficient way.

$$[ChildModelMatrix] = [ParentModelMatrix][Translation][Rotation][Scale]$$

4.1 Amillary Core

In the project, parent-child transforms are applied to the rotating Amillary and its rings. The core of the sphere uses the position of the gem at the centre of the guardian statue to rotate around it guardian keeping a constant circular position at every time.

- GGP = Guardian's Gem Position

$$[AmillaryCoreModelMatrix] = [GGP][TBTranslation][TBRotation][Scale]$$

The position is also further propagated to a spotlight, that follows the core of the amillary all along its circular journey.

Listing 3: Amillary Core transformations

```
    // Rotates the amillary around the statue
    float amillaryRotation = sin(rotationAngle) * 0.01f;

    // Selects the displacement position on Y and Z, and ↩
slows down the rotation speed
    auto rotationPosition = vec3(cos(rotationAngle)*155.0f, ↩
0, sin(rotationAngle * 0.35f)*155.0f);

    // Rotates the core on itself
    meshes["amillary"].get_transform().rotate(vec3(↩
amillaryRotation, amillaryRotation, amillaryRotation));

    // Moves the core around the statue
    meshes["amillary"].get_transform().position = ↩
rotationPosition + gemPosition;

    // Propagates the position to the spotlight
    points[5].set_position(meshes["amillary"].get_transform↩
().position);


```

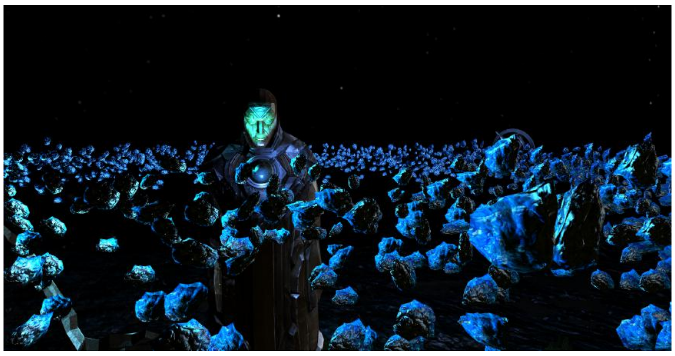

Figure 9: **Guardians' Gem** -The fulcrum of the amillary rotation

4.2 Hierarchical Rings

So far, the transforms are only based on the position of the parent, not very interesting. Things get a lot trickier with the rotating rings of the Amillary. Each ring Rn does instead calculates its rotation from Rn-1, and each one of then moved into position and rotated according to the rotation of an other common ancestor: the core of the amillary. This leads to a seamless and smooth transformation and rotational hierarchy where each ring fits well into its parent and rotates accordingly.

$$[M] = [PersonalRotation]x[PersonalScaling]$$

$$[RNModel] = [AmillaryCoreTransforms]x[AncestorsRotations]x[M]$$

The final result, is a smooth looking and rotating amillary where the rings follow and rotate around the emissive core in a flawless and seamless fashon.

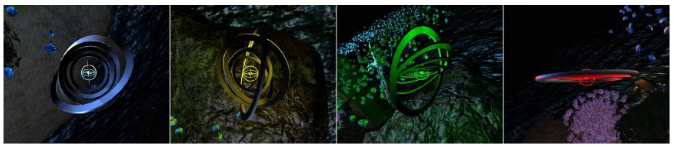

Figure 10: **Rotating Amillary** - The final result of the rotating amillary. Each ring spins magically on its own follows the core around the statue. The different colorus show different stages of the rotation

5 Normal-Mapping

Normal-Mapping is a technique used to simulate the complexity of bumps, dents and different heights found within a given surfaced. The simulation is used in computer graphics to achieve a good looking results when recreating the look and texture of a material while saving the computational power that would be used to render the uneven forms of the surface through polygons. Generally speaking a normal map is often a simple RGP projection of the object or a texture that can be used in conjuction with a normal, tangent a binormal matrix to reacreate the desired look of the material or object.

5.1 Calculating a normal map

To generate a normal map and apply to a mesh working with the tangent space is necessary. This is is based on the normal at a particular point of the object, the calculation is made using tangent, normal and binormal of the surface as the axes in our coordinate space.

In order to achieve the result we calculate the matrix:

$$TBN = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_x & B_x & N_z \end{bmatrix}$$

The values are then transformed based on the normal in order to generate the new normal which is used for all further calculations.

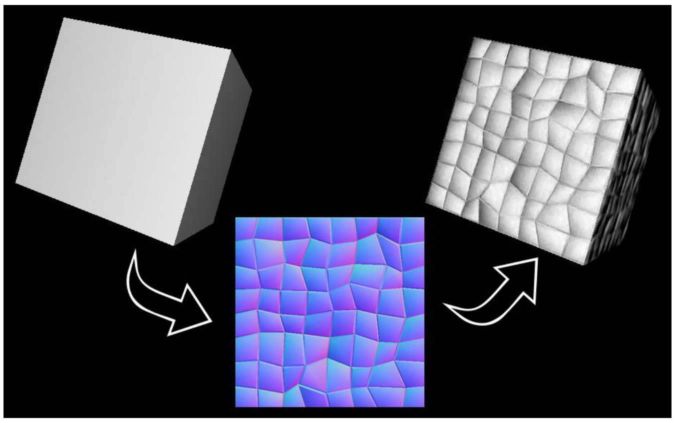

Figure 11: **Normal Mapping** - The usage of a normal map to texture a cube

Listing 4: uses the TBN matrix to calculate the new normal

```

        // returns the new normal
        return normalize(TBN * sampled_normal);

```

Most of the meshes rendered inside the scene do make use of normal mapping unless a decent good looking normal map was not available.

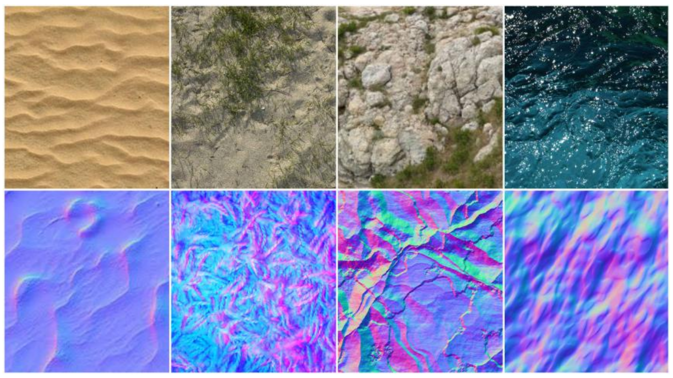

Figure 12: **Respective textures and normal maps for the scene** - Sand, Grass/Send, Overgrown Rocks and water

6 Terrain Generation

‘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.‘[Jerome Guinot 2008]

Hero’s Rest does include two separate height maps, one of the terrain and one for the water.

6.1 Terrain

Terrain the scene is the result of loading and parsing the height map of an island. The parsed map is used to generate a polygonal 3D representation of the image. The image itself, is then processed again in order to obtain normal, tanget and binormal for the surface of the generated terrain. The displaced vertices are then processed one last time in order to calculate a displacement on the Y-Axis used to calculate the weighting of each texture, this allows the rendering of a different texture (and normal map) and blend them based on the y value of a vertex, or in other words its height in the scene. This kind of technique allows us to render multiple different textures based on the position of the surface within the scene and to recreate the effect of a sandy shore normal-mapped with a sandy texture blending into one covered with short green grass.

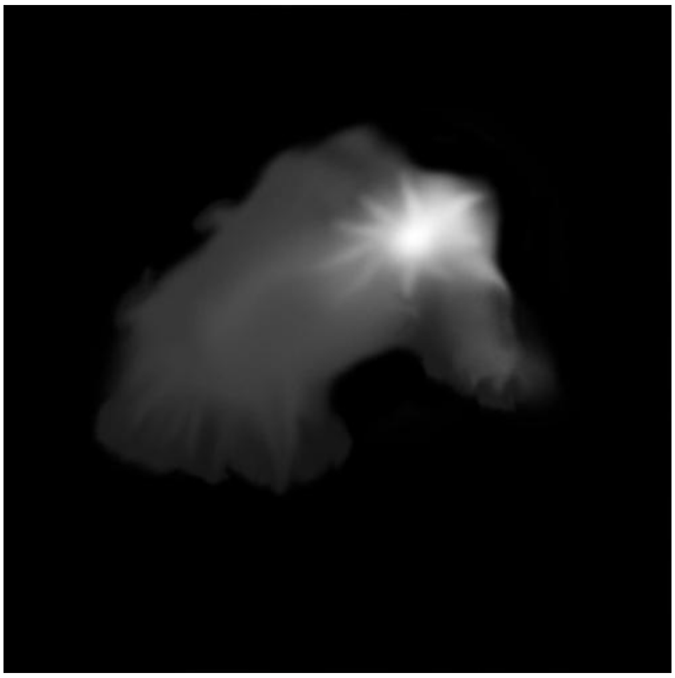

Figure 13: **- Island Height-Mapt** - Terrain

6.2 Water

Similarly to the terrain, water is generated through a height map, the main difference is that most of the height for the displacement on the Y-Axis is reduced for the water in the attempt to provide a wavy, yet less bumpy polygonal map. Moreover, the water presented in the scene does not use weighting and blending, but simply renders one single texture and one normal map. These two do however posses scrolling coordinates and textures in order to simulate the movement of the water in the sea. ‘...we never step into the same river twice...‘ [Heraclitus]

The two planes are finally combined together through simple geometric translations to provide the desired effect.

7 Skybox

Each scene would be incredibly computationally intensive to render if its dimensions went on for incredibly long distances. In order to solve the problem and provide a re-

Figure 14: **Final Result** - Seen from the tree

alistic looking sky, horizon and distance view to the settings skyboxes are often implemented. In the scene, the starry sky horizon, is at its core a simple set of six textures arranged as a cube map. These are seamless textures which are position in the scene on the faces of a clockwise rendered cube that encircles the graphical scene. The further scaled skybox, is then rendered with a dis-

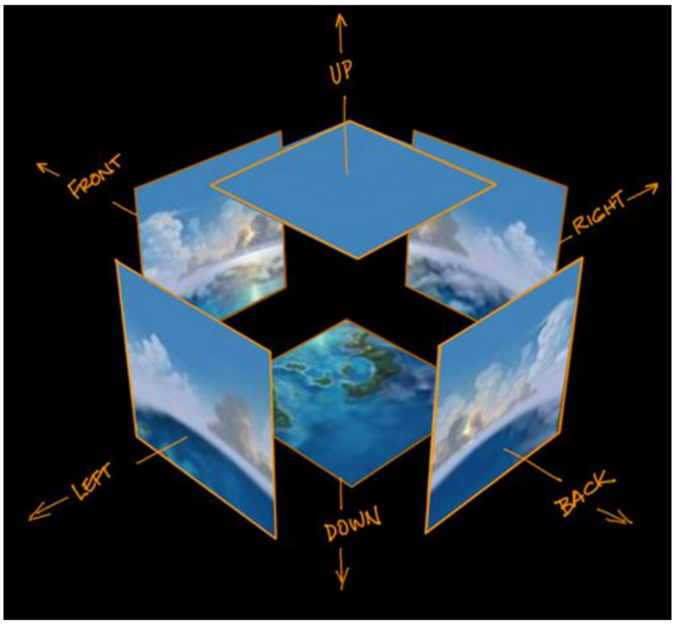

Figure 15: **- A skybox cube functionality**

abled depth buffer in order to offer the illusion of stillness when a camera moves around the scene. The final result is the idea of an incredibly large scenario, far from the user that spans over the horizon over the rendered scene all for the cost of a simple cube.

8 Shadow Maps

Shadow Maps are one of the various techniques used to create realistic looking shadows for the meshes rendered in a given scene. The technique revolves around the idea of capturing the information regarding the shadow projected by a spot light into a depth buffer. This is done by rendering the scene in two passes. The first one, gen-

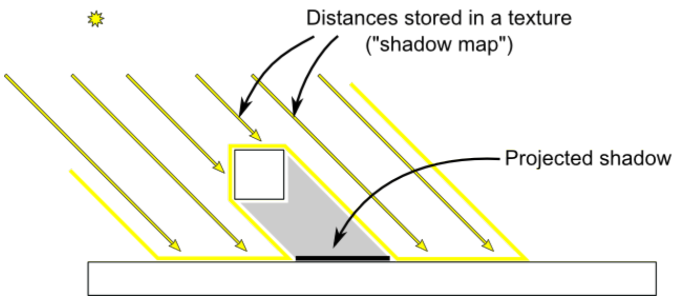

Figure 16: **- Object casting a shadow based on a spotlight**

erates the shadow map by rendering the scene from the light perspective and capturing the information, later to be used to actually render the shadows. The depth buffer containing the shadow information calculated using direction and position of a light source, are then used for the second pass. The second pass use shadow map to check whether or not a given fragment is in shade or if it is exposed to the specific light source that generated the shadow in the first place.

8.1 Generating the shadow map

In order to fill the buffer, it is necessary to calculate the scene from the perspective of the light, and thus to create a light screen space coordinate using the lightMVP for each one of the meshes.

Listing 5: generates lightMVP for each one of the meshesl

```
        // generates the light MVP matrix
        auto lightMVP = lightProjectionMatrix * V * M;

```

On the second pass, the same MVP matrix is then used not to render the meshes, but instead to calculate whether or not they affected by the shadow casted by the objects obstructing the bound light source, and thus shading their projection, sampling the depth buffer generated during the first pass based on the coordinates of the fragment and the lightMVP matrix. The result, is finally used to

Figure 17: **Shadows** - A generated shadow map for the tree

darken the color of the texture where necessary.

9 Future

Working with graphics many interesting features seen in games do come to mind as interesting options to implement. In future iterations of the projects, so far there a few ideas that are worth investigating for further iterations of the project.

9.1 Better Shadows

Shadow maps are currently available in the projects, but sadly some of the equations used suffer from issues related to the resolution of the map decaying over distance. Their implementation in a large scene such as Hero's Rest caues unwanted graphical artificats and overall a poor graphic quality. This is likely the first thing that will need further research before a new version of the project

9.2 Dynamically Generated Grass

Textures are nice, but definitively there are better ways to make terrain look good. One of this is to cover an area with green sprouty looking grass. This could be achievable by generating clusters of quads at certain positions, and clipping out the unwanted background through alpha mapping. The use of geometry shaders is however necessary, and this will require further investigation. [?]

9.3 Rain

I games atmospheric effects are often an incredibly nice touch to make the setting less repetitive and feel more alive. The island and the maelstrom would probably look a lot nicer under pouring rain.

9.4 Oren-Nayar shading

The phong model is a nice starting option, there are however better models for rendering Matte objects. Especially for things like the wooden boat, the ground or the guardian statue it might be very much worth to investigate these, such as Oren-Nayar shading.

10 Conclusion

[?] In conclusion the final result of the project was overall pretty successful even if maybe a bit unevenly.

The implementation of multiple shading models for different objects offers a rather distinguished look between meshes. The water with its scrolling textures looks realistic and calm, while the ground offers a detailed and overall familiar look with a blend of different textures. The various transforms make the rotating amillary and the debris look alive and realistic as if a real wizard was actually using the island for its spell and offer a great looking towering contrast with the Samurai's Grave and it's sharp Katana. On the bad side, while the moon looks great up in the sky, the shadows rendered from her spotlight beam are far from realistic. Point lights are also not taken in account in the overall calculations. Moreover, many of the surfaces suffer by common issues of being rendered by a Phong shader, and often seem a bit too shiny for what they are.

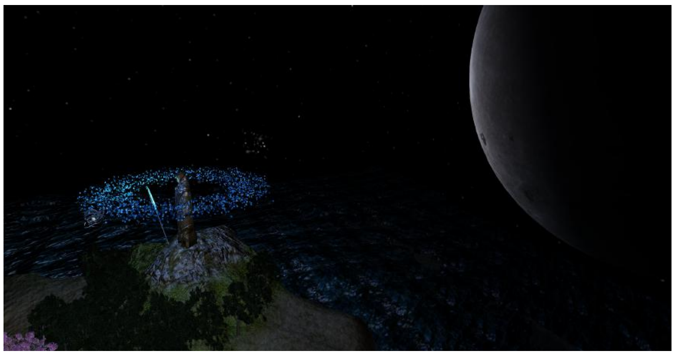
Figure 18: **Final Result** - Seen from the moon mesh

Bottom line, for a first iteration of the project the visual are a success until one begins too look too closely at some of the details. As always in programming, there no limit to how now can improve a scene, but I truly believe that many of the currently less successful features will be much better in the next implementation of the project.