Chance Island

Marcel K. Troscianko *
Edinburgh Napier University
Computer Graphics (SET08116)

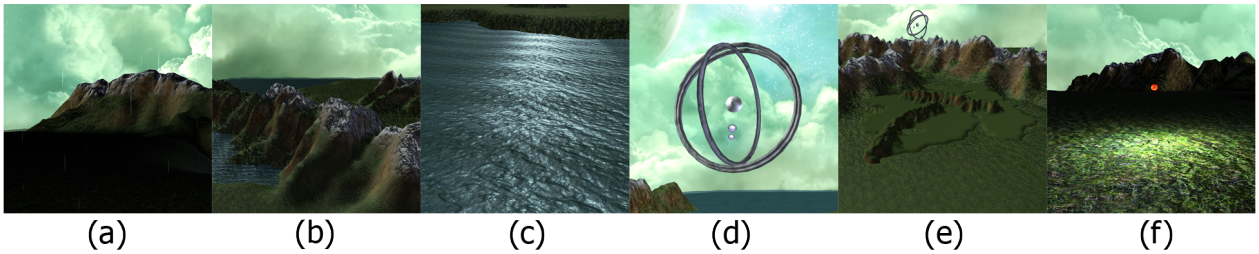

Figure 1: *(a) Rain effect, (b) Procedurally generated terrain, (c) Water geometry displacement, (d) Hierarchal movement, (e) Minimap, (f) Hierarchal lighting*

Abstract

This project was created to show off the ability to create a fresh, realistic looking scene each time it is switched on through random generation, but still paying attention to the detail many others would miss. For example, the water has two normal maps that move in different directions which, coupled with it's geometry displacement, creates a real feeling of depth in the water which manages to feel so alive during every sunrise and sunset. Perhaps the most interesting part of the project is the fully implemented hierarchy which manifests itself in the centre of the map as a mesmerising Gimbal-sphere relation.

Keywords: generation, normal, displacement, hierarchy, lighting, rain

1 Introduction

This project uses multiple effects to create a beautiful world every time it is generated. Both Gouraud and Phong lighting are used and implemented in a day-night cycle, and can be switched between at any time, creating a great testing environment for the effects that are used. There is a multitude of environmental effects such as water displacement via geometry shader, overlaying multiple normal maps and textures, and distorting post-processing on the rain.

All the major features will be covered within this report in detail, and extra ones will be shown in the recorded video test.

2 Related Work

Except for the workbook, the only piece of work that really inspired me was the ENBSeries' rain effect for Skyrim, developed by Boris Vorontsov

3 Overview

This scene creates a really unique look for itself through the use of it's more advanced effects and beautiful lighting which are all tied together by it;s solid hierarchy. Still, the very foundations of this world lie in it's random generation and the ability to always surprise with a new look.

*e-mail:29938203@napier.ac.uk

4 Methods/Techniques

This section will go into the details of the more technical parts of the coursework, explaining the mathetmatics behind the heavier calculations and the logic used for the less obvious effects. The parts of the implementation that will be discussed are:

- Lighting - Gouraud / Phong and the parts of a lighting equation
- Normal Mapping
- Procedural Generation - Basics of this randomly generated world
- Skybox
- Hierarchy
- Frustum Culling
- Rain - Generated on the geometry shader, and with post-processing
- Gaussian Blur
- Greyscale / Sepia
- Edge Detection
- Cel Chading
- Motion Blur

4.1 Lighting

The project uses two styles of lighting shaders - Gouraud and Phong. Both of them work in the same way, the difference being which part of the graphics pipeline they execute their commands in. These lighting calculations must be applied to some form of light source, of which there are three main ones: directional, point, and spot lights. Directional light is constant within one direction based on the normals of an object. Point light is similar, but it's direction must be calculated from the light's source point and the point it's shining upon. Spot lights can be thought of as a mix between the two - it originates from a point, but only flows within a a set range of directions.

Those lights are composed of the same two basic components; diffuse and specular lighting. Directional lights also have an ambient light property, but that is a constant value. Diffuse lighting is the main component, as it is the one that lights up the object purely based on it's angle to the light source (see Figure 13). The specular lighting is the spec of light that you see when light reflects directly into your eyes from a highly reflective area of the object. The ambient component is the easiest to calculate, as it's just a constant value. The diffuse is the next on the list, and is calculated as shown in Equation 1.

$$diffuse = max(normal \cdot light_{dir}, 0) \times material_c \times light_c \quad (1)$$

Where $normal$ is the normal to the surface of the material at that point, $material_c$ is the material colour, and $light_c$ is the light colour. Specular light is a bit more tricky to calculate as it requires a few more values. We start off by calculating the $half\_vector$ as demonstrated in Equation 2

$$half\_vector = normalize(light_{dir} + view_{dir}) \quad (2)$$

$$direction_{view} = normalize(eye_{pos} - object_{pos}) \quad (3)$$

Where l is again the light, and $direction_v$ is the view direction which is calculated as in Equation 3. With this ready, we can calculate the specular position as in Equation 4 and Equation 5

$$k = max(normal \cdot half\_vector)^{material\_shininess} \quad (4)$$

$$specular = k * material_{spec\_colour} * light_c \quad (5)$$

Where $light_c$ is the colour of the light. This gives us all we need to calculate directional light at a point (see Equation 6).

$$colour = (material_e + diffue + ambient) * tex\_colour + specular \quad (6)$$

Where tex_colour is the sampled texture, $ambient$ is a light constant, and $material_e$ is the emissive (glow) value of a material.

4.2 Normal Mapping

Normal mapping is used extensively within the project - every visible object besides the skybox has a normal map applied to it. The obvious downside of applying normal maps to objects is the increase of the visual gap between the Gouraud and Phong shaders, which is caused by the texture being sampled within the fragment shader and so becoming unusable with the Gouraud calculations;

Applying a normal map to an object is simple - the normal map is sampled, and the RGD values are saved as a new normal X, Y, and Z co-ordinates. We also need the binormals and tangents of the object to transform our original normal. To do this, we align them with a pre-existing axis (X and Z axis). This is just done as a cross product between the normal and a vector (0, 0, 1) for the binormal, and (1, 0, 0) for the tangent.

To get apply this to our normal, we set a 3x3 matrix using our old normal and newly calculated binormal and tangent (Equation 7), then we multiply that by the sampled normal (Equation 8).

$$normal\_matrix = (normal, binormal, tangent) \quad (7)$$

$$new\_normal = normalize(normal\_matrix * sampled\_normal) \quad (8)$$

This project goes one step further and applies two normal maps onto the ocean, moving in different directions. Both normal maps are sampled, using different texture coordinates which are calculated on the geometry shader, along with the water displacement. The two resulting normals are then mixed into a final normal which consists uses 70% of the first normal and 30% of the second normal. This gives the water a more realistic three dimensional effect.

4.3 Procedural Generation

This project doesn't use procedural generation based on an image heightmap, but instead randomly generates it's own. This is done in a number of steps that will be explain below.

1. A flat heightmap is passed into a seed generator along with a number of seeds that it must plant. A seed is simply a single point which it raises to something above zero. While the seeder is doing this, it also generates a direction map for the seeds - every point above zero on the heightmap has a number between zero and one assigned to it on the direction map. This number will indicate whether this piece of land is more likely to expand width-wise or depth-wise.

2. Once the world is seeded, the heightmap is passed into a land generator. This goes through every point on the heightmap and checks whether or not it has any land points immediately around it (the eight surrounding points). Points which are above or below increase the depth counter, points which are to either side increase the width counter, and points which are in the corners increase the generic counter. A direction counter is also kept to see what values the direction map has stored for every surrounding point. If the direction counter has more width points and the width counter is high, the point has a higher chance of becoming a piece of land, and the same is true for the depth version of this. This creates a flat continent. This is run through a number of multi-threaded passes.

3. Once the land generator is done, the program passes into the mountain seeder. Mountains seed high points, and tend to follow the last trend. This means that if the last point fell, it's more likely for this point to also fall. If the last point rose, it's likely for this point to also rise. The same is true for changing the x-z direction. This generates snake-like shapes in the terrain. At any point in this generation, there is a small chance that the mountain range will spawn a child mountain range, which will split off and expand on its own. It's usually best to put very few seeds into the mountains, because their seeder expands a lot.

4. When the mountain seeder has done its job, the mountain generator takes over. It is similar to the land generator, except it doesn't use direction maps and it starts at a high height value which is then decreased for every pass of the generator. This, again, is run through a number of passes.

5. Finally, the terrain is run through a smoothing function which takes a user-specified smoothing range, then sets each point to

that points' height plus the average of the surrounding points over two. This is described in Equation 9:

$$height = (height + average\_surrounding\_height)/2 \tag{9}$$

6. In the terrain fragment shader, the texture is chosen based on the height (grass/snow) and on the angle of the slope (rock)

Figure 2 shows an example of the randomly generated world

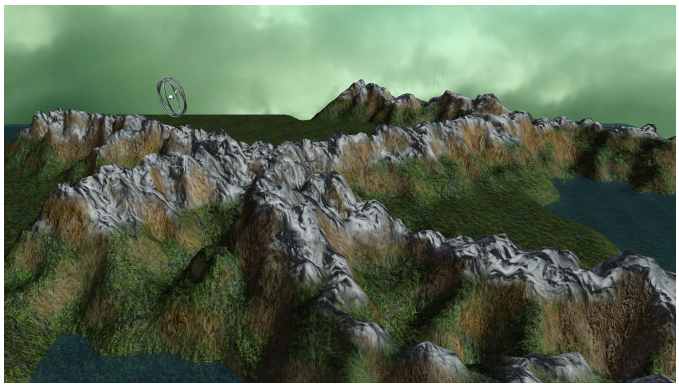

Figure 2: *Randomly generated terrain*

4.4 Skybox

Adding a skybox is a great way to add atmosphere to the scene - it adds a seemingly huge space for the player to be in, while not costing much processing power. The technique is to create an inversed cube so that it is visible from the inside, then apply 6 textures onto it, one per side - this is called a cube map. The depth buffer is disabled as the skybox is rendered which means that any future models get rendered over it no matter how far away they are. This makes the skybox seem a lot bigger than it actually is. Just like any other texture, the skybox can be animated or blended to allow for weather effects and a day and night cycle.

This project uses the default skybox supplied by the tutorials.

4.5 Hierarchy

A parent-child hierarchy was implemented to allow for hierarchal transformations. The design for this was losely based on a binary tree - A node class named $scene\_object$ was created to hold a mesh, any textures the mesh uses, and an effect. To allow for parent-child hierarchy, every node also contains an array of pointers to it's children, as well as a world transformation matrix which contains the true transformation of the node in relation to all of it's parents. The world transformation matrix is implemented as shown in Equations 10, 11 and 12 below.

$$world\_transform = (\prod_{p=0}^{n} transform_p) \times transform_{local} \tag{10}$$

$$transform_p = translation_p \times scale_p \times rotation_p \tag{11}$$

$$transform_{local} = translation_{local} \times scale_{local} \times rotation_{local} \tag{12}$$

Where p is the parent node, $local$ is the current node, and n is the number of parent nodes.
This allows us to find the true transformation of the object in with respect to it's parents. That alone isn't enough information for rendering, however - we also need to know by how much we need to transform the node's normal matrix to stay true to it's hierarchal transformation. The way this is found is extremely similar to the above method, and is shown in Equation 13 below.

$$world\_normal = (\prod_{p=0}^{n} normal\_matrix_p) \times normal\_matrix_{local} \tag{13}$$

Where, again, p is the parent node and $local$ is the current node.
The final part of the hierarchy is linking lights to scene objects. This is done by simply multiplying their position by the parent's world transform matrix when we bind it to a shader.

Showing this off to it's full potential in the program can be a challenge, so I've chosen to use a gimbal with scaling spheres (see Figure 3), as these use a multi-layered hierarchy and all angles of rotation.

Figure 3: *Gimbal with spheres*

To show lighting in hierarchy, the program has a point light bound to a sphere spinning around the camera (Figure 4));

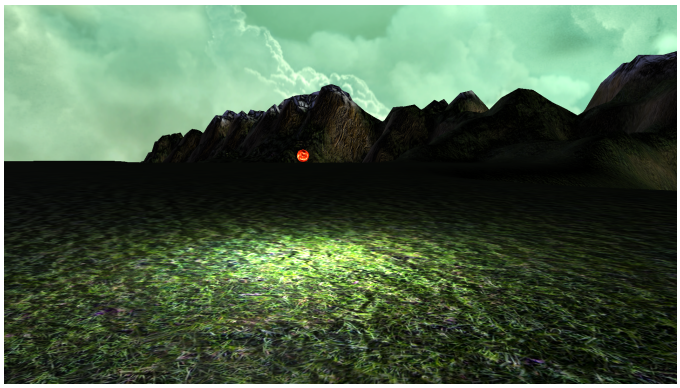

Figure 4: *Point light attached to parent Sphere*

4.6 Frustum Culling

Frustum Culling is a technique used to greatly increase performance by not rendering object that aren't within the View Frustum - they aren't going to show up anyway but, depending on how many objects you have, may seriously impact the program's performance. The way to do this is to find the six planes of the frustum, then not render any object that is on the back of any of the plane. Equation 14 shows how to find if a point is behind or on a plane.

$$is\_in\_front = dot(object\_point - plane\_point, plane\_normal) > 0 \quad (14)$$

where $plane\_point$ is any point on the plane.

Calculating the points on each of the frustum's plane only requires we find the four corner points of the near and far plane - the other planes share some of these points since it's a six-sided shape - and calculating the normals for each of these planes only requires three of the points. We manually set the distance to the near and far planes, the aspect ratio, and the field of view of the camera. The normal to the near plane is just the normalised camera look direction ($cam.get\_target() - cam.get\_position()$), and the normal to the far plane is the inverse of that. So we have the distances to each of the planes, the normals to each of the planes, the camera field of view, and the camera aspect rati0. We will also need the camera x-axis direction, but all we need to find that is do the cross product of the camera y-axis direction and the camera look direction ($cross(cam.get\_up(), (cam.get\_target() - cam.get\_position())$). All we need to find the four points on each of the planes now is to find their centre points (from there, we can simply use the fov and aspect ratio to find the corner points). Finding the centre is also very easy - it's just the normalised camera look direction multiplied by the distance (near plane distance for near plane centre, far plane distance for far plane centre). And now we have all we need for frustum culling!

4.7 Rain

This project doesn't use blend maps as such, but it does use generated blend maps as a way to perform post-processing on the rain. The goal of this process is to give rain a thicker, more liquid feel. To achieve this, the project first renders the entire scene onto a frame buffer as it normally would to the screen. The rain is rendered next - the geometry shader receives only lines, indicating the position of the start and end of a rain drop, and converts them into billboard rectangles. Doing this is quite simple, because we set the camera position as a uniform in the main.cpp file. The geometry shader finds the midpoint of the rain drop ($top\_point - bottom\_point$), then gets the vector between that and the camera. This gives us enough information to work out the local x-axis, which is just a cross product of the normal and the vector between the bottom point and the top point of the raindrop. This is used to create the rectangle, and apply texture co-ordinates onto it. This information is passed into the fragment shader.

The texture that is set for rain is just a greyscale image that goes from white on the outside to black on the inside. This is converted into an alpha map for the rain drop, with an alpha of 0.0 on the outside and 1.0 on the inside. This is then rendered into a separate frame buffer.

The final part of the operation is to combine the rain and terrain frame buffers. To do this, a screen space quad is set (a quad with co-ordinates from (-1, -1, 0) to (1, 1, 0)) and the terrain and rain frame buffers are passed as textures into the renderer. When the terrain texture is applied in the fragment shader, it is moved upwards by 0.03 times the strength of the colour of the rain frame buffer, which creates a distortion based on how close that point is to the centre of a rain drop. An extra darkening is applied onto every raindrop to make it look more realistic.

Figure 5 Shows the normal rain effect, and Figure 6 Shows it with post-processing enabled

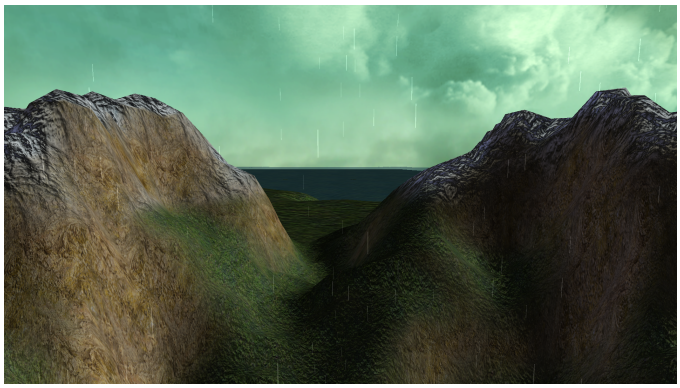

Figure 5: *Normal Rain*

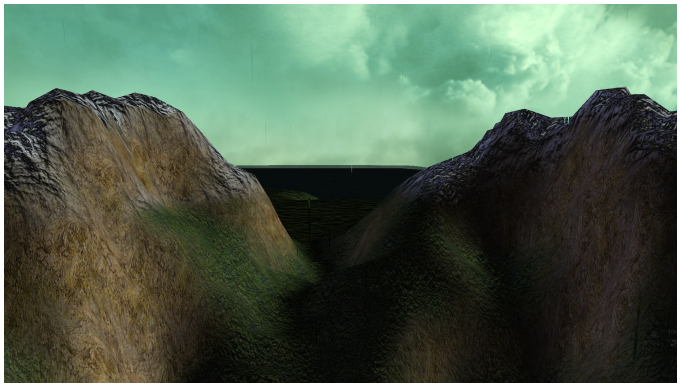

Figure 6: *Post-Processing Rain*

4.8 Gaussian Blur

This is a multiple pass effect designed to blur an image. Bluring an image alone isn't a very useful effect at all, but it's often used as a step in other effects like bloom and depth-of-field. Gaussian Blur blurs an image by setting each pixel's colour to the average of it's surrounding pixels in a set range, and each surrounding pixel's influence depends on it's distance to the centre pixel. This can be performed in one of two ways - each pixel can either check a square of pixels around it in one go, or the image can be blurred vertically and horizontally in separate passes. The latter is much faster, as you only check two lines per blur, as opposed to a full square. The amount of operations per pass of a square blur can be calculated as in Equation 15

$$num\_operations_{square} = (range * 2 + 1)^2 \quad (15)$$

Whereas the amount of operations per pass of a line blur can be calculated as in Equation 16

$$num\_operations_{line} = 2 * (range * 2 + 1) \quad (16)$$

This clearly shows that the processing power for a square blur increases much faster than for a line blur as the range increases. It should be noted, however, that for a small range a square blur may be less efficient because a line blur requires two render passes. The actual equation for a horizontal line blur is shown in Equation 17 below:

$$pix\_colour = \sum_{n=-range}^{range} pix\_colour_{tex\_coords+(n,0)} * (1 - \frac{|n|}{range}) \tag{17}$$

For the verticle blur you just chance (n, 0) to (0, n).

4.9 Bloom

Bloom is an effect that immitates the spread of light off light sources and brightly lit surfaces. The clear first step in this is to get the brightest part of the image, because that's what our light sources/brightest surfaces are. To immitate spread, the image is then blurred a few times using the Gaussian Blur effect above, then placed over the original image. Equation 18 shows how to get the brightest spots of an image using a cutoff brightness.

$$colour = (i > cutoff)?vec4(i, i, i, 1.0) : 0.0 \tag{18}$$

Where i = intensity. Intensity itself can be found in a number of ways, the most popular being to take the red, green, and blue components of the image, then dividing that by three to get a scale from 0 to 1. This is a fairly good attempty, but it means that really bright lights of individual colours (bright red, for example) may not bloom. A fix for this is to set the intensity to whatever the brightest component of the pixel is instead, as shown in Equation 19 below.

$$intensity = max(max(pixel_r, pixel_g), pixel_b) \tag{19}$$

4.10 Greyscale / Sepia

Greyscale is about the simplest effect that can be used, but it can be done in two ways. An easy way of doing it is to get an average of the three colour components ($intensity = \frac{red+green+blue}{3}$) then setting that as the new three colour components ($vec3(intensity, intensity, intensity, 1.0)$). This is a fine rough estimate, but it doesn't really work - RGB isn't a perfect way of representing colour, and our sun isn't white but rather green, both of which mean that giving the same weight to each component ($\frac{1}{3}$) isn't going to give us an accurate output colour. Instead, it is recommended to add the intensity weights as $red = 0.299, green = 0.587, blue = 0.184$ according to ITU-R Recommendation BT.601, which is easily done by simply setting the intensity to the dot product of the a vector containing the intensities and that containing the colour, as shown in Equation 20

$$intensity = dot(pixel\_colour, (0.299, 0.587, 0.184, 0.0)) \tag{20}$$

Original:

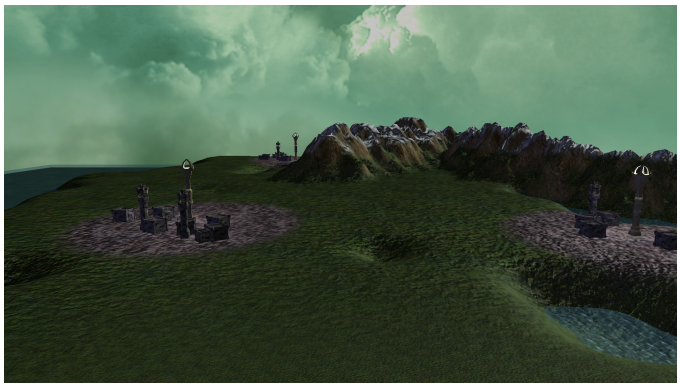

Figure 7: *Original image*

Effect:

Figure 8: *Image with Greyscale applied*

Sepia is an extremely similar effect - in fact, it requires Greyscale. The idea of Sepia is to take the Greyscale version of an image, then take add a sepia colour vector to it ($sepia = (red = 0.314, green = 0.169, blue = -0.090)$)

Effect:

Figure 9: *Greyscale image with Sepia applied*

Note: Another popular colour balance is $red = 0.2126, green = 0.7152, blue = 0.0722$, ITU-R Recommendation BT.709

4.11 Edge Detection

Edge detection is a technique of finding edges in images based on the difference of colour between the the surroundings of an image. To do this, the program checks the pixels at a specified range away from the centre pixel, and if the difference in colour intensity ((red + green + blue) / 3) is above a specified threshold, the program returns that difference. Otherwise, the program returns a 0 (black).

The range is 1 (checks only the pixels directly around the centre pixel), but increasing that range causes more edges to be detected. The reason for this is that gradients no longer act as smoothly as before. This technique is used in Cel Shading. Output of pure Edge Detection is shown in Figure 10

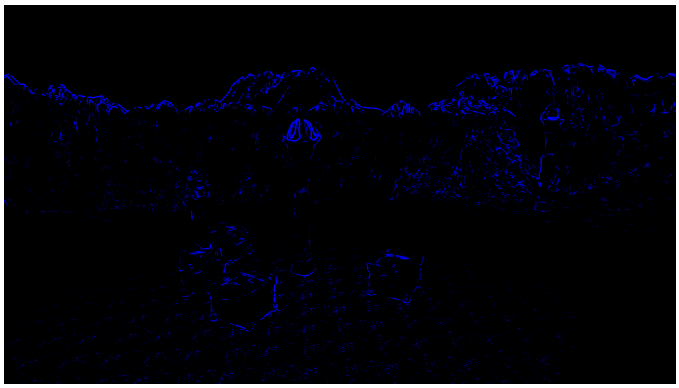

Figure 10: *Output screen from Edge Detection (Edges shown in blue)*

4.12 Cel Shading

Cel Shading alone is a very simple effect - Edge Detection checks for any edges and saves that processed image into a variable, image_edges. image_edges and the screen image, image_screen, are sent into the Cel Shader. Where the colour of image_edges is black, the shader outputs the colour of image_screen, and the shader outputs black where the colour of image_edges is anything other than black. Figure 11 shows the scene with Cel Shading on.

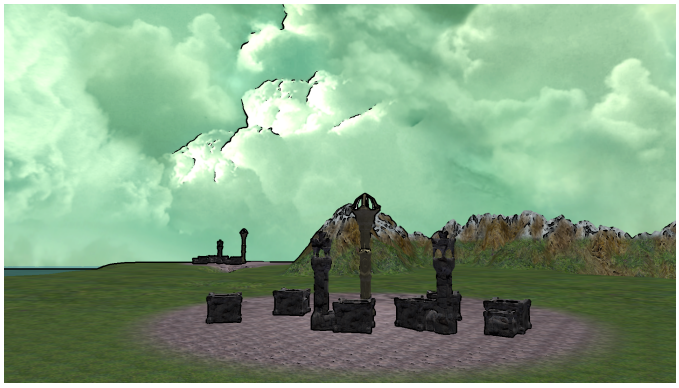

Figure 11: *Output from Cel Shading. Note how the it's similar to the effect of Edge Detection)*

4.13 Motion Blur

In real life you don't see things jumping straight from one position to another - they transition smoothly, and this is where Motion Blur comes in. The idea behind it is that you'll take the first current frame of the image, and apply a number of previous frames over the top at a very low opacity. This creates an a feeling of smooth transitioning. This is the second-easiest effect to implement, right after the two Greyscale effects. For this effect, the program stores the last few frames and a marker to the current one, then blends them in reverse order (for example, if there are 5 frames and we're currently writing to frame 3, we'd apply $(frame3 + 0.5 * (frame4 + 0.5 * (frame5 + 0.5 * (frame1 + 0.5 * (frame2)))))$, which will blend them together while making them all fade out the older they are). The effects of Motion Blur are shown in figure 12, but it can be hard to see without zooming in.

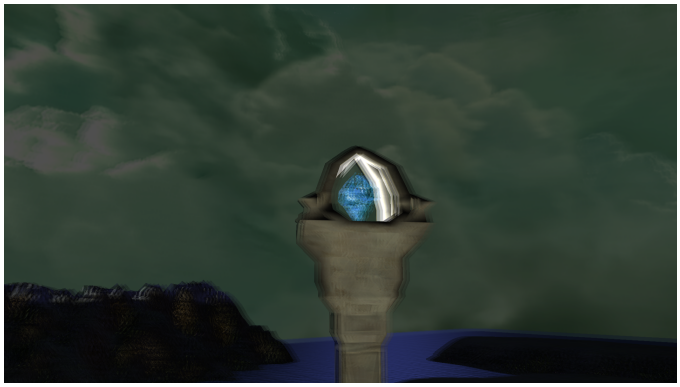

Figure 12: *Output from Motion Blur post-blend*

5 Conclusion

Putting great resolution scaling on the terrain and multi-layered normal mapping on the sea really brings out the shapes cast by every day in this island created literally by chance, which is perhaps it's most impressive feat - where most scenes stand still every time you see them, this one not only changes constantly while you're in it, but it will also never look the same twice. Adding the post-processing on the rain and toggle-able effects was a great choice because it really improves the exploration of this world and lets the user see each part of it in turn. That and the ominous spinning rings and spheres above the mountain tops make this whole scene feel very different and surreal.

6 Appendix

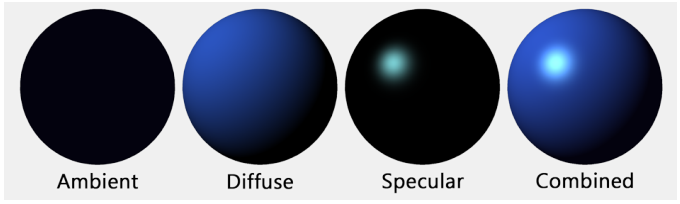

Figure 13: *Parts of a light*