POST-PROCESS EFFECTS IN GRAPHIC ENGINE

Tianshi Xie

Troy University

Troy, United States

txie@troy.edu

Abstract

This article describes how to design the post-processing system and take advantage of this system to implement the post-processing effects in 3D graphic engine**[51]**. This paper is divided into two parts. The first one is to create a 3D world because that is precondition of the post-processing system. The second one, to implement motion blur and water rendering by using the post-processing system. My contribution is that utilizing the 3D technology to deal with the real problem in the project and to optimize the performance from the rendering system and logic system. This paper involves a lot of rendering technology. Even you don't know them, don't worry about that since I am going to explain all of the important bits as I go along with that.

Keywords: Shader, Multitexturing, Matrices, Lighting, Fog, Mipmapping, Cel Shading, Normal Mapping, Shadow Mapping, Antialiasing, Lens Flare, Low-POLY.

- Shader : It is a computer program that is used to do shading.
- RenderTarget : It is a feature of modern graphics processing units (GPUs) that allows a 3D scene to be rendered to an intermediate memory buffer.
- Multi-pass rendering: It is a process in which an application traverses its scene graph multiple times in order to produce an output to render to the display.
- RenderMonkey: It is a software for rending system and developed by AMD company.
- Screen effect: Some graphics technique that is applied to the entire screen.

Contents

Introduction

Post-processing effects is used in Video and Film industries for improving the quality of the image (image processing methods). It is also widely used in real-time 3D rendering such as video games to add special effects (Anti-Aliasing, Bloom, Color Grading, Depth of Field, Lens Flare, Screen Space Reflections and so on). Besides that, the post-processing effects is very important in the simulation environment such as self-driving, the medical business and so on.

The process of post-processing effects is that drawing the scene to a buffer in the memory of the video card rather than rendering the graphic directly to the display and then using Pixel shader and Vertex shader to apply some filters to achieve some special visual effects.

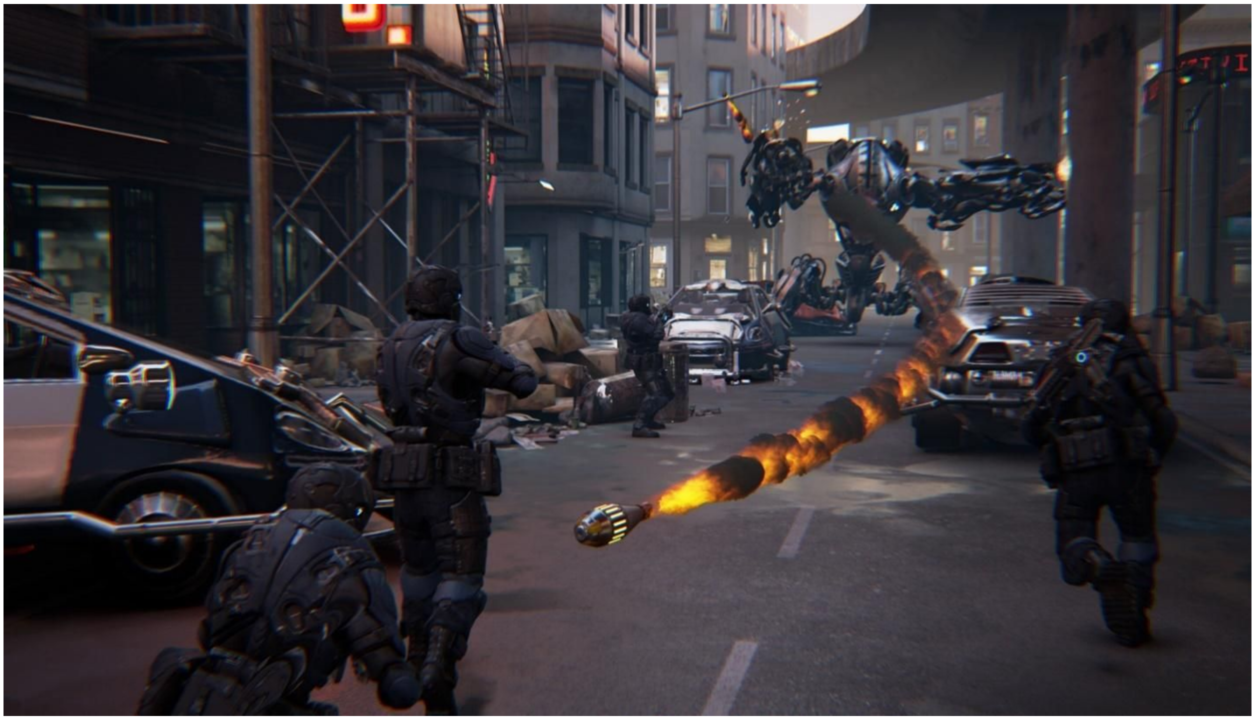


Figure 1

Figure 1 show that applying a post-processing effects a flying bullet in a video game.

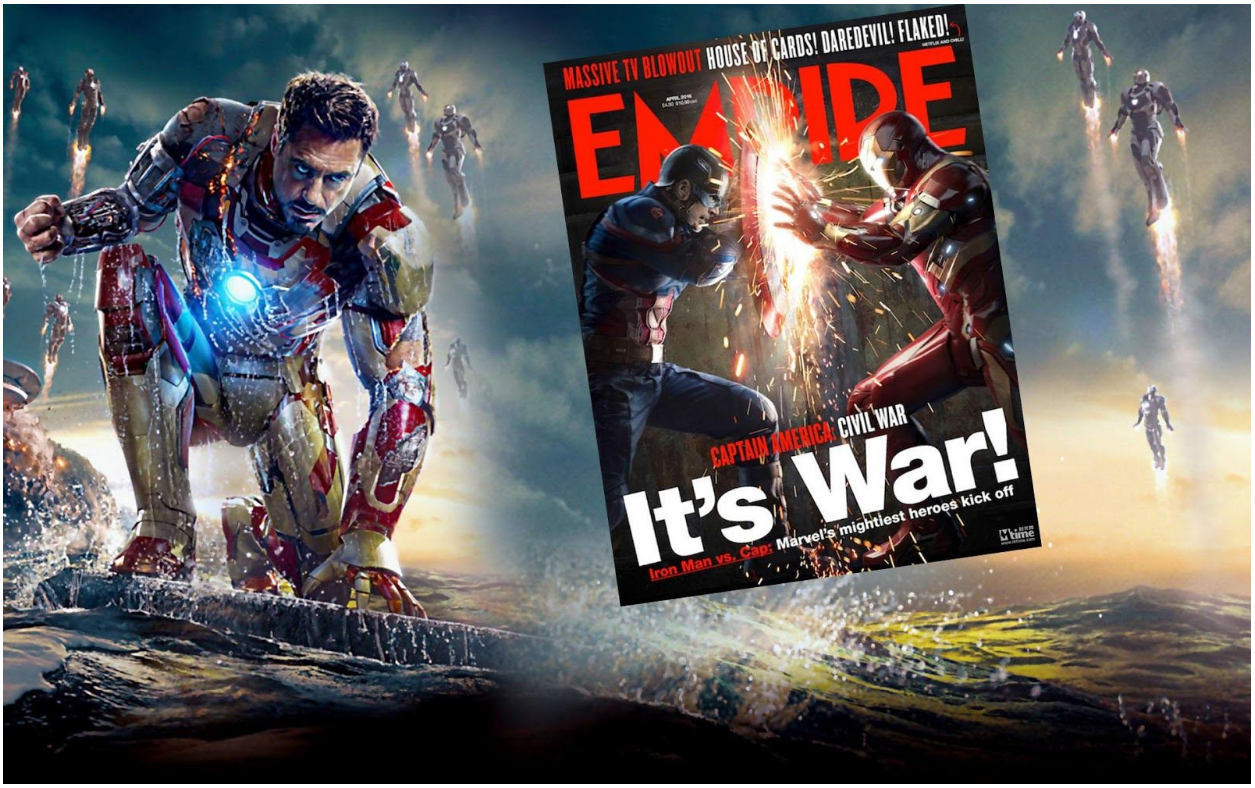

MASSIVE TV BLOWOUT HOUSE OF CARDS! DAREDEVIL! FLAKED!
EMPIRE
CAPTAIN AMERICA: CIVIL WAR
It's War!
Iron Man vs. Cap: Marvel's mightiest heroes kick off

Figure 2

Figure 2 is shown that applying the post-processing effects in a film. In this picture, a blue light went out of the iron man’s chest. When the iron man is touching the shield, sparks are flying. Their clothing textures can reflect different lighting effects. How to reach that effects? Post-processing effects is the key to solve these technology problems.

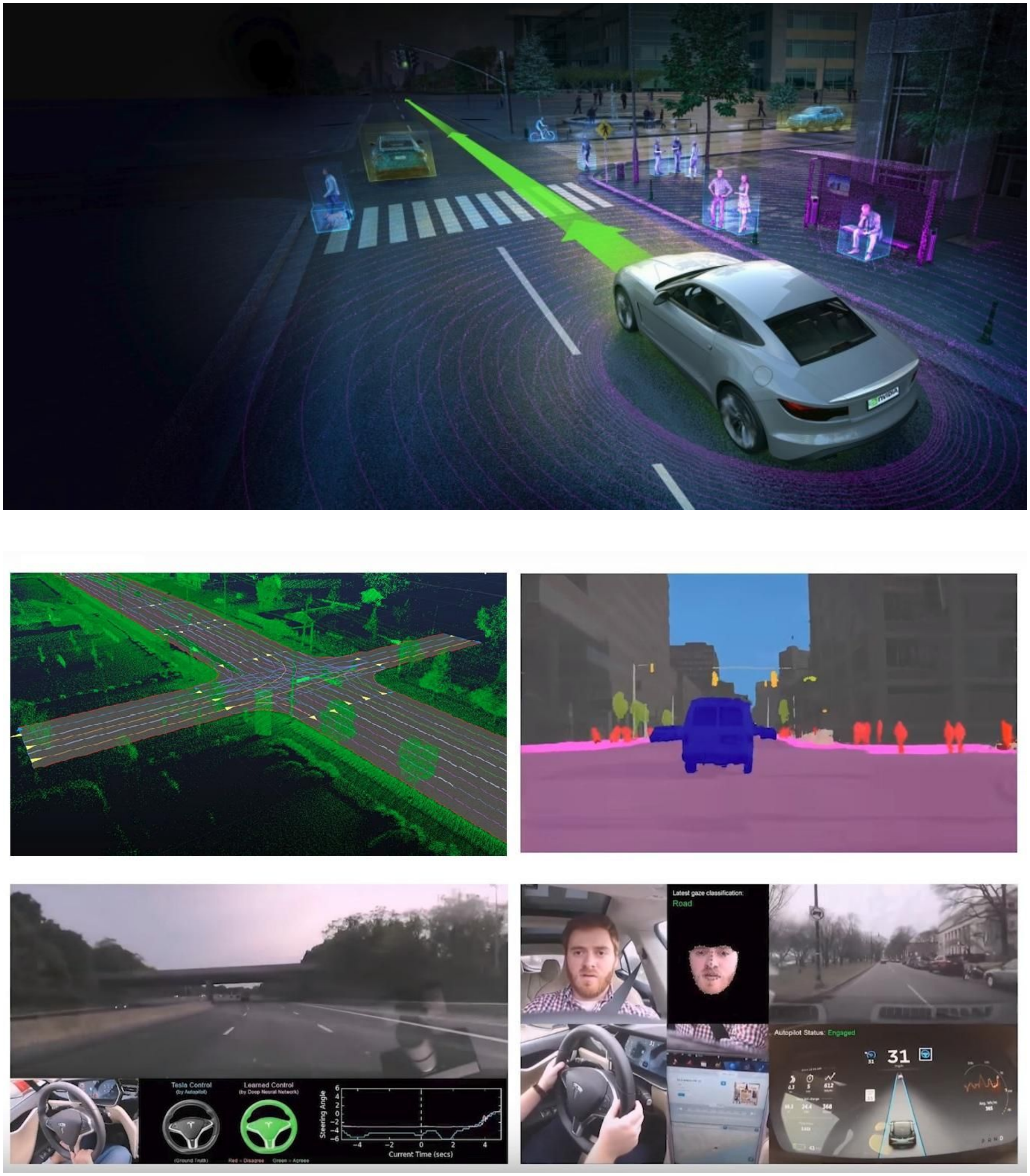

Tesla Control
(by Autopilot)
(Ground Truth)
Learned Control
(by Deep Neural Network)
Red = Disagree
Green = Agreee
Steering Angle
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
Current Time (secs)
Latest gaze classification:
Road
Autopilot Status: Engaged
31

Figure 3

Nowadays the machine learning is very hot, especially in self-driving. Before testing the real car on the road, testing the car in a simulation environment is necessary because that can make sure that the car is safe on the real road. The most important technology in a simulation environment is the post-processing effects that belong to the image processing method since it can filter the most important information for not only the machine learning system but also the user. For example, the post-processing effects can label and color the important objects in a scene so that users can analyze each situation clearly when the car is running and do not need to consider other unnecessary information.

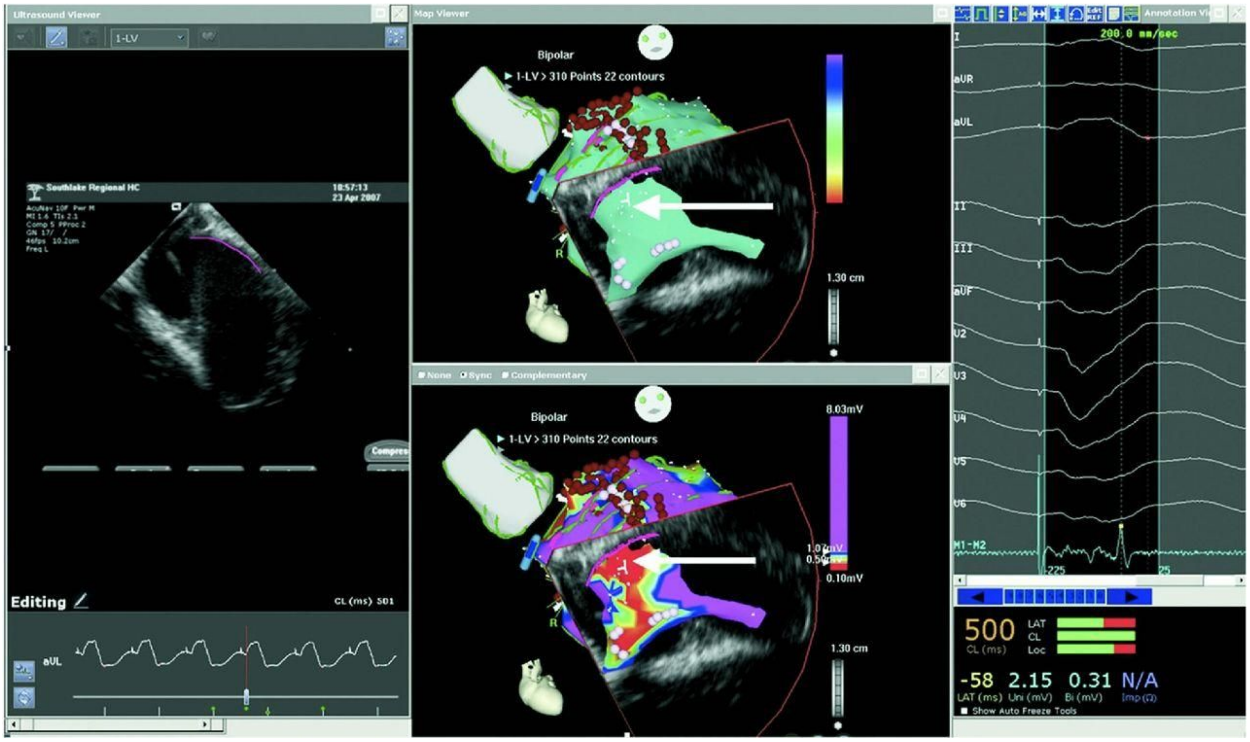
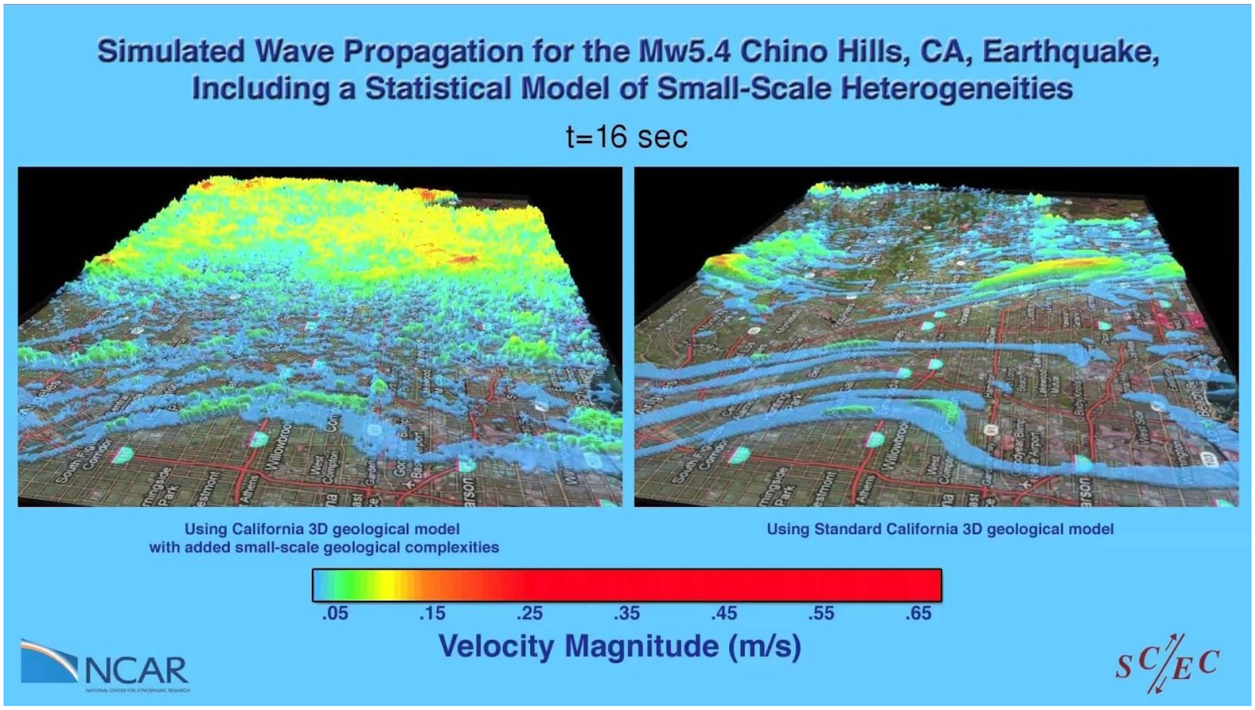


Figure 4



Simulated Wave Propagation for the Mw5.4 Chino Hills, CA, Earthquake, Including a Statistical Model of Small-Scale Heterogeneities

t=16 sec

Using California 3D geological model
with added small-scale geological complexities

Using Standard California 3D geological model

Figure 5

The post-processing effects also occupy an important place in the medical system and seismic system. The most important information can be presented different visual effects by all kinds of image processing algorithm.

Performance analysis

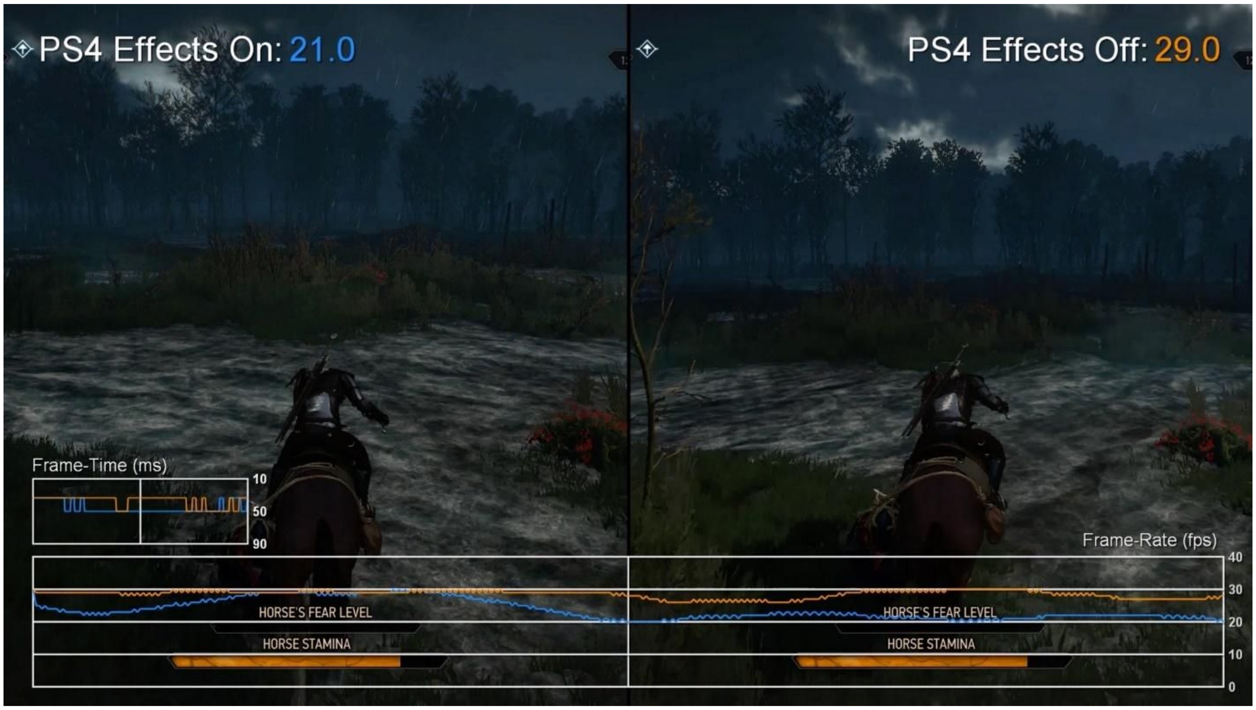


Figure 6

It is important to understand that post-processing always involves a trade-off between speed and image quality when enabling the post-processing system. Figure 6 show that is a very famous video game << The Witcher 3: Wild Hunt>>. When the post-processing system is turned on, the FPS(Frames Per Second) is 21. Instead, it is 29. Opening the switch of the post-processing means that to require more computation amount and memory space to obtain better visual effects. With the evolution of hardware, the post-processing effects become more and more complex since the demand of people is improving. So, the only way we can do is that reducing the time cost of the algorithm of post-processing effect in GPU, optimizing the data structure which is going to be passing into the memory of video card and cutting back the number of multiple-passes. In this project, I take advantage the MRT technology to reduce the number of multiple-passes for better performance.

Basic Rendering Unit

Rending a quad using vertex array objects and vertex buffer objects. Vertex array objects are the modern way of storing and rendering models.

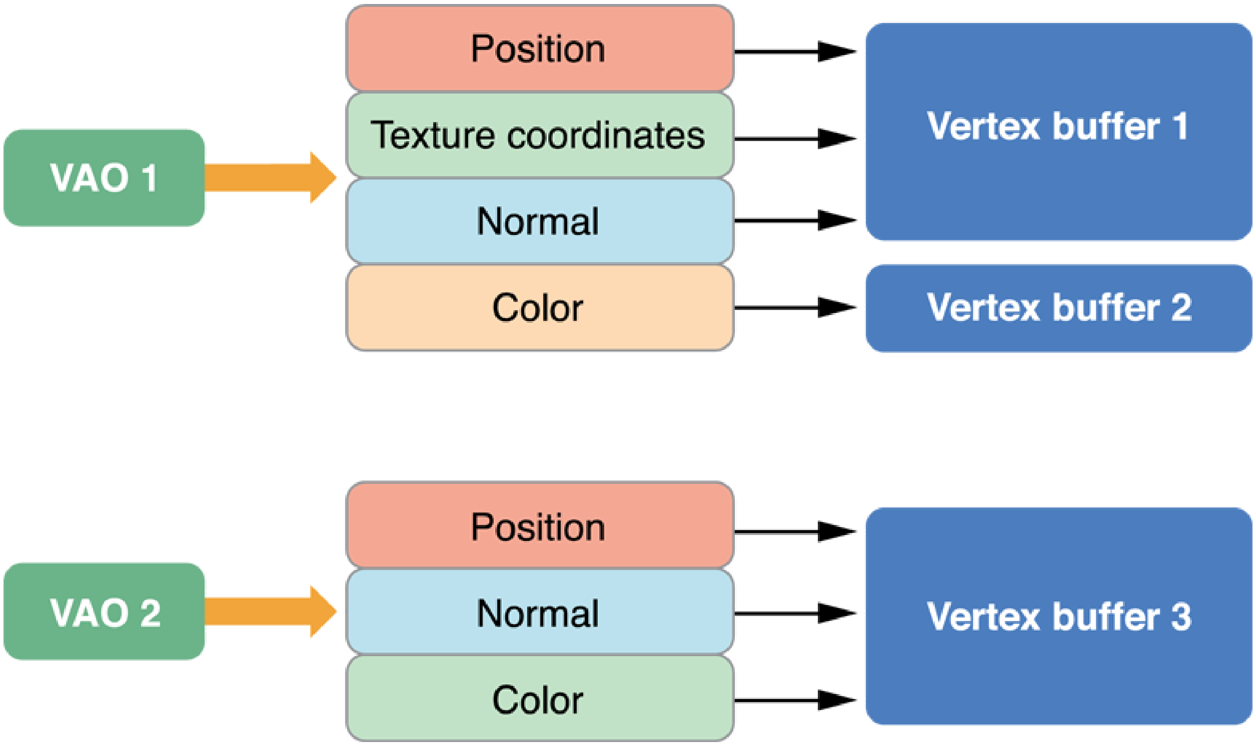


Figure 1.2: This picture shows the construction of VAO and VBO (Image courtesy of Apple Corp.)

Vertex Array Objects

A vertex array object of VAO is I am going to call them is an object in which you can store data about a 3d model the VAO has a load of slots in which you can store data and these slots are known as attribute lists and I think that a VAO usually has around 16 attribute lists. I am not going to need anything like that many. Anyway, but usually you store different sets of data into these attribute lists so in one you might store all the vertex positions in another you could stall the colors of all the vertices or the normal vector at each vertex or the texture coordinates. It’s up to you what data you store where and these data sets are stored in the attribute lists as vertex buffer objects

Vertex Buffer Objects

What’s a VBO. A VBO is just data you can see it as pretty much just an array of numbers very simple and this data could be anything it could be positions colors normal or whatever and each VBO can be put into a separate attribute list in the VAO so that’s how data is stored but how do we access this data whenever we need it each VAO has a unique ID so when the VAO is stored off in memory we can access it at any time using its ID but how exactly do we represent a 3d model as a load of data that we could store in a VAO. well every single 3d model that dealing with is made out of a number of triangles and each of these triangles has of course three vertices. Three points in 3-dimensional space so each vertex has a 3d coordinate an X Y and a Z and if we take the coordinates of each of three vertices of the triangle and then all of the coordinates from the other triangles in the model. We will have a list of data that represents all the vertex positions of the model and it’s this sort of data that we can put into the VBO and store into an attribute list of a VAO and that’s what is doing here.

Rendering the simplest of 3d models a rectangle which is made of just two triangles and taking the vertex positional data of those triangles creating a VBO. Storing the data into the VBO. Creating a VAO storing the VBO data into one of the attribute lists of the VAO and then we will use the ID of the VAO to tell OpenGL to render the VAO rectangle.

Related Work

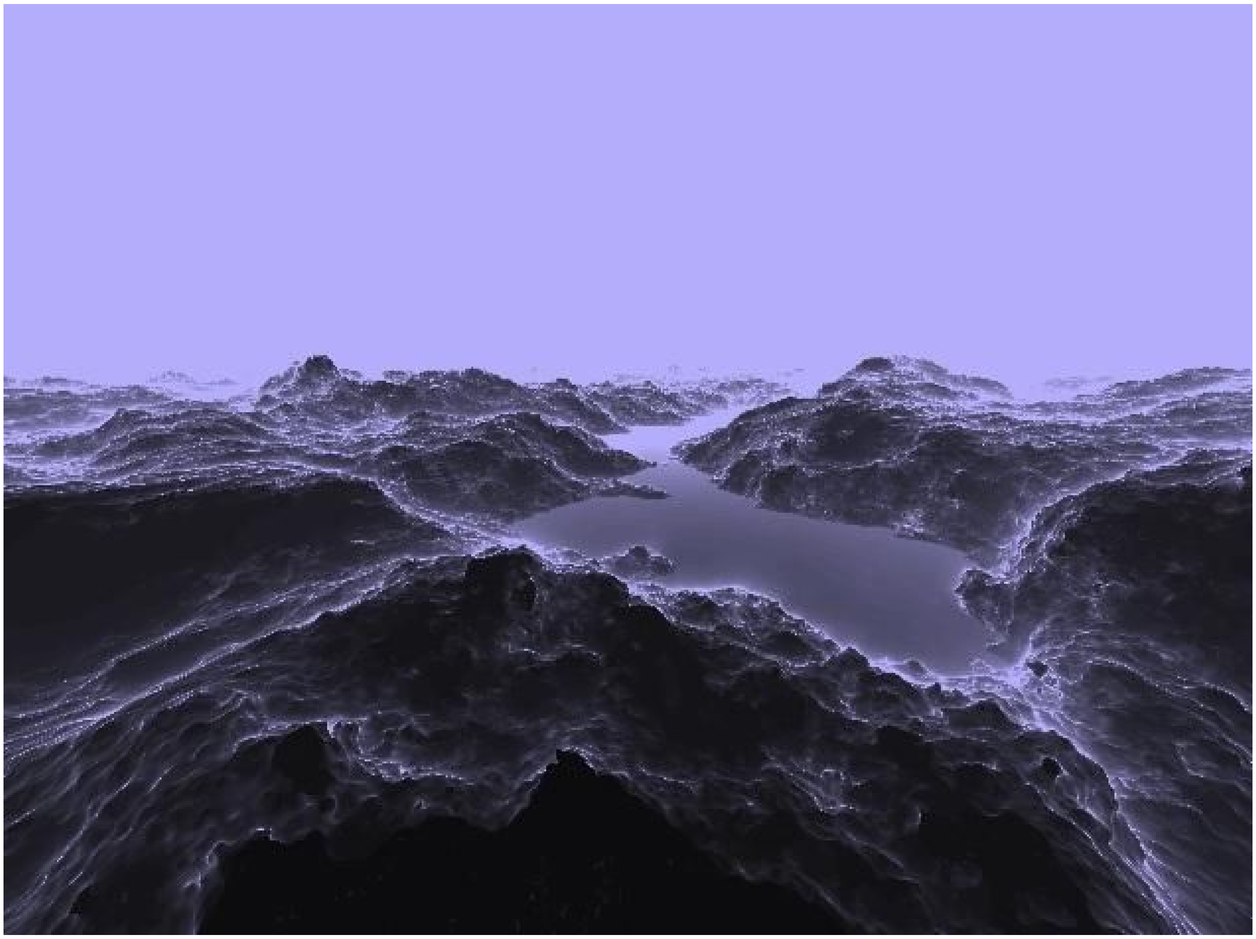
Creating A Scene

Here is to be creating a terrain rendering process and rendering a flat terrain so a terrain is actually pretty similar to any other entity. It’s just made out of a model which is a mesh[38] of vertices[39] and a texture[40] but the difference is that a terrain has a very regular layout of vertices making it relatively easy to generate here instead of having to create it in some 3D modeling software[41] our terrains are always going to be set out like this made out of a regular grid of vertices the vertices are always spaced out equally along the horizontal axis so from above the terrain it will always look like this then to add some hills or mountains or any other kind of heights to the terrain we just have to give each vertex its own individual height. There are many techniques that we can use to generate the heights of each vertex.

Now keeping it simple and set all the heights to 0 producing a completely flat terrain. The world is then going to be made up of a grid of terrain tiles like this one here. All of the terrains will be the same size and they will all have the same number of vertices along each edge of the terrain, each terrain has its own position in the grid which determines the terrains position in the world.

Terrain Generation

Taking advantage of the Perlin noise[42] algorithm to be implementing procedural terrain generation. That is a nice algorithm which can create some very nice-looking random terrains.



Lighting

Without lighting, you can't see anything in the world. Adding lighting into the world can make everything looks vivid.

Diffuse Lighting

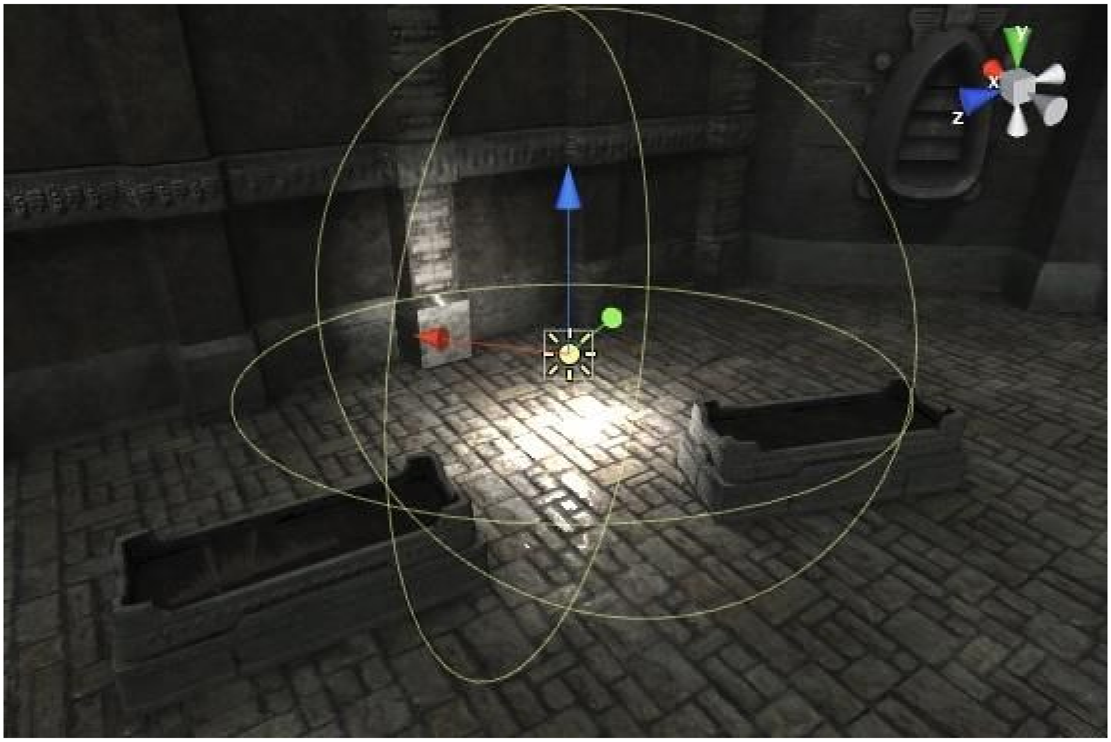
All it needs really is a position so a vector[43] 3f a 3d coordinate for the position and also a vector 3f for the color which is also going to be like the light intensity. The lighting calculations are all going to be carried out in the shader program so that need to have access to the position and color of the light in our shader so in the vertex shader to set up a vec3 called light position which is going to obviously hold the position of the light and then in the fragment shader I am going to need another vector and this is going to be the light color. So now we need to do the usual things with the uniform variables so in the static shader class we need two more integer which are going to hold the location of the light position uniform variable and the light color uniform variable just as we did for all the matrices and as usual we have to get the location of the uniforms in the get all uniform locations method so we do that once for the variable which was called like position which was in the vertex shader and we will do the same again saving this in location light color. I am going to save the location of the variable that was called light color and now I need a method to load up these light variable so I am going to create a method called load light. It's going to take in a light and then to make use of the load vector method in the shader program class to load up the light position to location light position and then we want to load up the light color to location light color and that will load up the values to the shaders so now we have the light position and color in the shader but we also need something else to allow us to calculate how bright a given point on an object should be if you have a look around you in your room or wherever you are I am sure that there will be some light coming from somewhere and some objects around you will presumably be being lit up by this light and if you these objects you will of course see that size of the object that are facing the light are more lit up than parts of the object that are facing away from the light so the more that the surface faces the light the brighter it appears so in order to do on lighting calculation. it will need some indication of which direction parts of the model are facing and this is where normal vectors come into play a normal vector is a direction vector that points in the exact direction that the surface every point on the surface has a normal vector and it's always pointing directly out of the face or in other words it's perpendicular to the face basically the normal of a surface tells us which direction it's facing in and that is exactly what we need and luckily for us the obj files provide us with the normal vector at each vertex of our model and we loaded all of these normal into a float array so if we want to be able to access these normal in the shader we need to put them into the models VAO[44] along with the vertex positions and the texture coordinates if you remember from before in our models VAO we currently store vertex position in attribute zero and texture coordinates in attribute 1 so we might as well go ahead and store the normal in attribute two.

Specular Lighting

The specular lighting[45] gives the objects a shiny look and feel so specular lighting is not something we use instead of diffuse lighting but we use it in addition to the diffuse lighting but what ‘s specular lighting specular lighting is reflected light on shiny objects and in many ways is basically just the reflection of the light source in the surface of the shiny object so let’s imagine that this here is the surface of a shiny object and that there’s a light source here let’s now work out what the specular lighting would be like at this point here which would have a surface normal like this light will come down in this direction from the light source and hit the vertex now not only will the point be lit up with diffuse lighting[46] that we calculated above but because this is a shiny surface some of the lights will be reflected off like this hopefully you can see how the reflected light corresponds to the incoming light ray and the surface normal here if you imagine the light source being here instead the light would be reflected like this or if the light source was here then obviously it would be reflected like this and so on the amount of light that’s reflected depends on the reflectivity of the surface so representing this in our code by giving our entities a reflectivity attribute if the object has a high reflectivity then the reflected light will be stronger and if it has a low reflectivity then the reflected light will be much weaker if the object has a reflectivity of zero there no light would be reflected at all and there would be no specular lighting the brightness of this point also depends on one final factor and that is the position of the camera. If the camera is positioned here then the point would have very high specular brightness because the reflected light is going straight into the camera if the camera is positioned somewhere like here then pretty much none of the reflected light would go into the camera and so the point wouldn’t appear any brighter than a completely non reflective surface it wouldn’t have any specular lighting at all finally if the camera was here it would probably get some of the light going into it and so the point would have a bit of specular lighting but it depends on the material so to have another attribute called shine dampening which determines how close the camera needs to be to the reflected light to see any change in the brightness on the surface of the objects.

Point Lighting

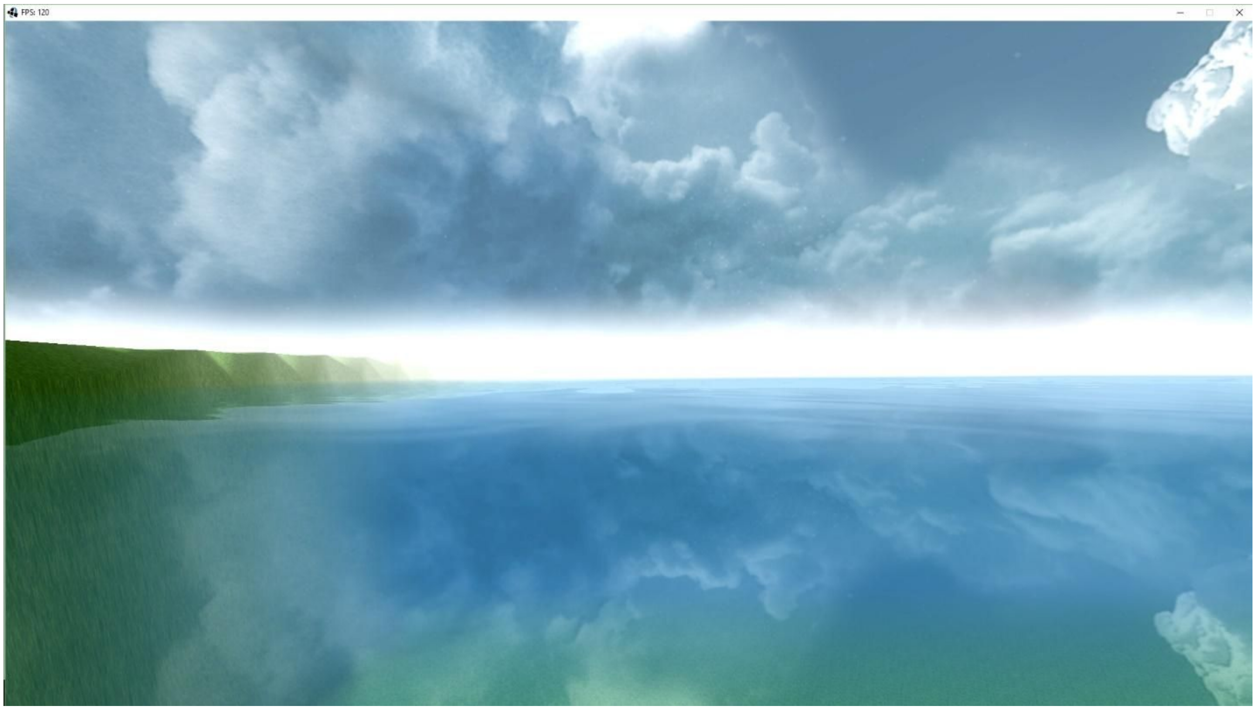
Having multiple light sources in the scene but the range of these lights is infinite and they light up the whole world which isn’t very realistic at all usually the lighting caused by a light gets dimmer the further away from the light source and this is what is known as attenuation**[47]** the lights brightness varies with distance so to be using this equation here to calculate the brightness of a light and as you can see it’s very simple obviously the larger the attenuation the less bright the light so we just need to calculate this attenuation factor and we can do that using this equation here so you can see here that as the distance from the light source increases so will the attenuation factor and the three attenuation coefficients here are values that we can choose allowing us to determine exactly how a certain light source attenuates.



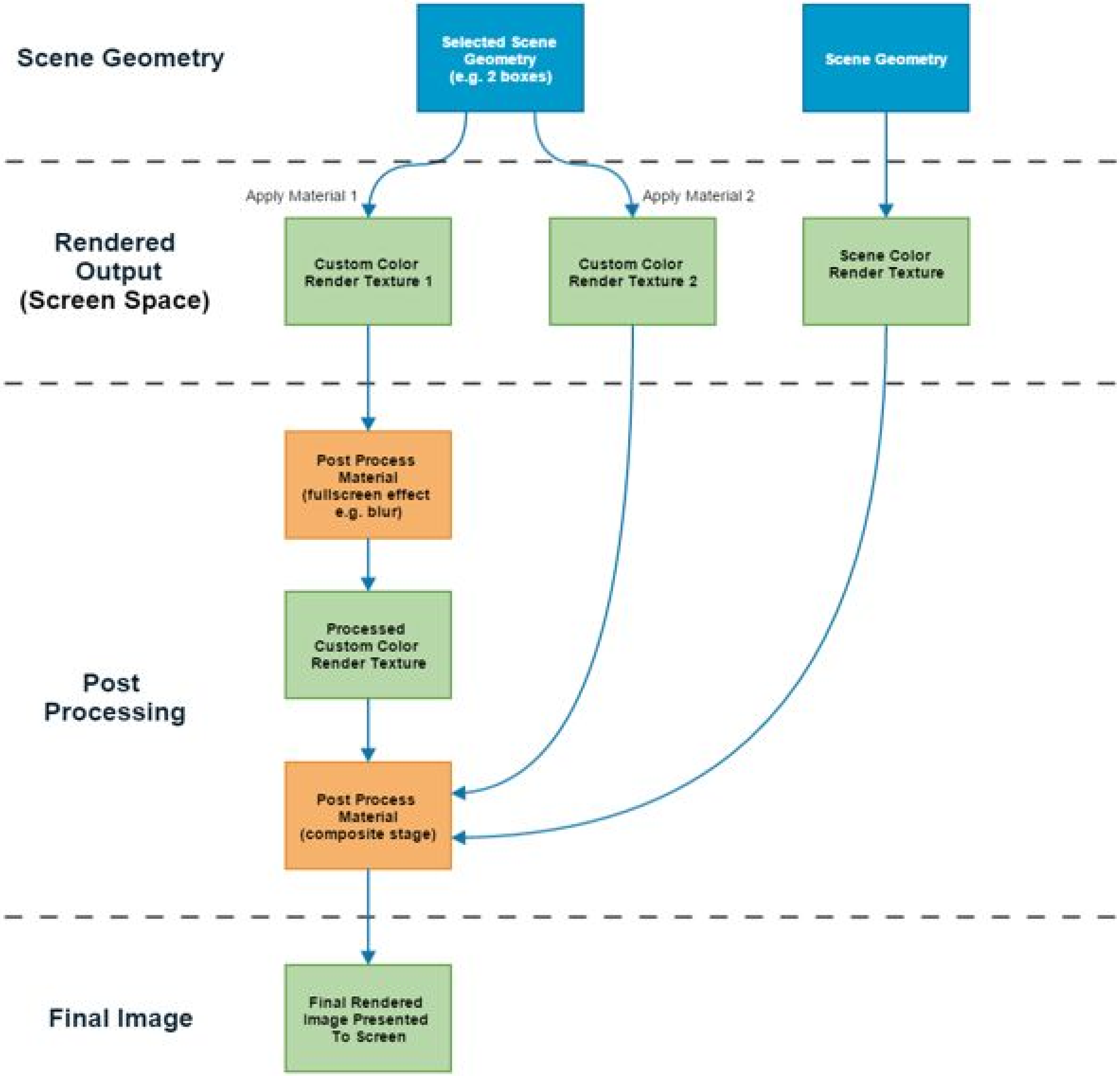
Now to add an attenuation for every light source and this is going to be a vector 3f because it’s going to be three value it’s not really a vector but it will make our lives easier if we store these three attenuation coefficients as a vector and I am going to set that as default to one zero zero because if you check the equations that will lead to no attenuation at all and I am also going to create an extra constructor here so that if we want we can choose to initialize a light with a set attenuation so we can actually set the attenuation for a light source and of course to need to have a getter method to get the attenuation so to be doing the calculation in the fragment shader so in the fragment shader we need to take in those three attenuation values so we will take in a uniform VEC3**[43]** attenuation and we need one for each light source and seeing as there are four light sources I am going to make that an array of size for so to do the attenuation calculation we need the distance of this particular fragment from the light source and this here this two light vector is a vector from the fragment to the light so if we get the length of that vector then we know the distance to the light source so we have to do this for every light we will get the distance to that particular light source and then we can do that attenuation calculation to calculate the attenuation factor and if you remember if you look back at the equation that is attenuation 1 or attenuation x in this case and its attenuation 1 because it's for each light source so attenuation 1 or attenuation x plus attenuation 2 multiplied by the distance plus attenuation 3 multiplied by the distance squared which is of course the same as distance multiplied by distance so we have calculated the attenuation factor and now we need to divide the lighting by that factor so we will do it for the diffuse lighting first so this here is the diffuse lighting for each light which we then add on to the total diffuse but we just need to divide that by the attenuation factor for this light and we need to do the same for the specular lighting for this light so that's all of this bit and we will divide that by the attenuation factor so that is all the attenuation calculation done so in the static shader we have to do the usual stuff same as above. We now need an array of locations for the attenuation because of course the attenuation is an array there is one for each light source then like above we have to initialize that array events. So we will create a new int array and the size if going to be max lights and then to use that to go into this for loop and get the location of all of the elements in that attenuation array and to make sure you change that to attenuation so now that we have got the location of all the elements in the attenuation array we can load up some values to them so in the load lights method we now need to load up the lights attenuation as well as the color and position so to location attenuation I and load up lights get a dot get attenuation and when we load up the empty lights we still need to load up some value so that it doesn’t try dividing by zero so to load up to location attenuation I to load up a new vector which is going to be one, zero, zero because we don’t want it to be dividing by zero in the shader. so at this point we have completely implemented attenuation for the entity shader but we now need to go back and do exactly the same thing for the terrain shader.

SkyBox

A skybox[48] is just a simple box with a nice texture of the sky all around it. The box contains all of the visible world and it follows the camera around meaning that the camera is always in the center of the box and it can never reach the edges of the skybox. Until now we have only ever used simple 2d textures like this one here but cube maps are textures in the shape of a 3d cube where each side of the cube is basically a 2d texture but this whole thing counts as just one texture, one texture with six sections cube maps can be used for a number things but obviously they are perfectly suited for creating a skybox instead of using six individual textures. To texture the skybox we can just use one cube map texture will create the cube map from six separate images but it will be stored as one texture this means that before we render the skybox we just have to bind that one cube map and then we can render the whole skybox one other thing that’s slightly special about the cube maps is the way that they have to be sampled usually to sample a texture we provide a 2D texture coordinate and the color of that point on the texture is returned. However this obviously won’t work with the cube map seeing as it’s not a simple 2d texture it’s a 3d cube so to sample a cube map we have to provide a 3D Direction vector the direction vector points from the center of the cube to the point on the texture that we want to sample for example a direction vector of 0 1 0 would point directly upwards and therefore would sample the pixel in the center of the top face a direction of 1 0 0 would sample the sends a pixel on the right face or a direction of minus one minus one minus one would sample the back left bottom corner so we can sample any point of the cube map by using the relevant direction vector and the best thing about this and the reason why cube maps are so perfect for sky boxes is that the vertex positions of a cube can also be used as the direction vectors for sampling the cube map so we won't be needing any texture coordinates and just create a cube model from a load of vertex positions and then those vertex positions will double up as the direction vectors used to sample the cube map in the fragment. I am going to use to make up this cube map so in the method the first thing is to generate a completely empty texture by calling GL gen textures and that returns the ID of that empty texture so that we can now use it the first thing is to bind that texture so that we can do stuff to it so that we can store data in it so we have activated texture unit 0 and now binding the texture to that texture unit by calling GL bind texture usually when we call this we put in GL texture 2d but this time as the first parameter we are putting in GL texture cube map to indicate to open jail that this texture is going to be a cube map so now looping through all of these six textures and loading up each one into the cube map so we will loop through them all and the first thing we need to do to each texture is to load it up into a texture data object so we have the data of that texture available to us so that we can load it into the cube map so it will be called by the decode texture file methods all of the textures are going to be in the res folder and they are all going to end in dot PNG[50] so in this array of strings that this method takes in we just need the actual names of the textures then load up the texture data into the cube map[49].



Post-Processing Effects


Scene Geometry
Selected Scene Geometry (e.g. 2 boxes)
Scene Geometry
Apply Material 1
Apply Material 2
Rendered Output (Screen Space)
Custom Color Render Texture 1
Custom Color Render Texture 2
Scene Color Render Texture
Post Process Material (fullscreen effect e.g. blur)
Processed Custom Color Render Texture
Post Processing
Post Process Material (composite stage)
Final Image
Final Rendered Image Presented To Screen

The basics of post-processing effects work by rendering the scene to a texture rather than straight to the screen so that you basically have a screen image of the 3d scene you can then process that 2d image in the same way that you processed images in programs like Photoshop.

Before actually rendering it onto the screen so the first step of post-processing is to render the scene to a texture and using Frame Buffer objects. so instead of rendering the 3d scene to the screen is to render it to an FBO and that will give us a 2D texture with an image the scene and also a 2d texture of the depth buffer if we need it to put this image onto the screen so that we can actually see it and render the FBO texture onto a 2d quad and this chord will fill up the entire screen this will be done in the exact same way that we render the 2d textured quads so at this stage all of this seems pretty pointless the project will still look exactly as it did before and we might as well have just rendered it straight to the screen rather than rendering it to a texture and then rendering that texture onto a chords on the screen however this now gives us the opportunity to carry out some post-processing effects on the image of the scene when we render that textured quad onto the screen we obviously have to use a shader program. when we rendered the texture, the shader program determines the color of every pixel on that quad so instead of just setting the color of the quad straight to the color of the texture. we can do something a little bit fancier so for a very simple example we could set the output color of the fragment shader to 1 minus the color of the texture therefore inverting all of the colors in the scene when they get rendered to the quads and that would be a very simple example of a post-processing effect also our post-processing paint plane doesn’t just have to have one step like in example here instead of rendering that full screen quad onto the screen we could instead render it to another FBO giving us a new texture that we use we can then render a textured quad with that image onto another FBO using a different shader program to apply a different effect such as a horizontal Gaussian blur and we can repeat this as many times as we want with different effects so i could then apply a vertical Gaussian blur and finally render that image onto a quad on the screen applying a shader program that changes the contrast of the image and it’s important to understand that is not rendering the whole scene each time here we only render the 3d scene once to the first FBO and then all of these extra steps just require a single texture quad to be rendered using a certain shader program to apply the effect.

Firstly, we can create a FBO of a certain size with a certain type of depth buffer. and then created a 2d quad which will fill up the whole display and before any post processing takes place the quads BAO is bound so that the quad can be rendered and when the post processing pipeline is finished the quad is unbound depth and testing is also disabled during post-processing, Only rendering a single quad each time so any post-processing that we want to occur will have to take place here between the start and end method calls in the processing method finally we will build a method for rendering the quads to either the screen or to another FBO. Now to create a post-processing effect which will alter the contrast of the image so to do this to render the 3d scene to an FBO and then we will need to render that image onto a textured quad on the screen using a shader program that changes the contrast of the image so the first thing that to do here is to create this step here which renders a textured quad onto the screen using a shader program that alters the contrast so let’s render that textured quad and use the shader program to change the contrast of the color in that quad so, to need an image renderer to render the quads and a shader program, next to make change the contrast then bind the texture and bind it to texture units so that can be sampled in the fragment shader and to use the render that quad with that texture and then we can stop the shader program once that’s happened next up we need to add the contrast changer effect to our post processing pipeline so that the contrast actually gets changed when we do the post-processing and of course we need to remember to release the resource after using that. when the post processing is carried out, carry out the contrast change effect and give it the color texture which will be an image of the scene and of course to get that image of the scene we need to have rendered all of our scene to an FBO so to create an FBO and make sure that everything that rendering to the scene at the moment actually gets rendered to the FBO instead so this FBO is going to be the same size as the screen we want it of course to be the same resolution and it is going to need a depth buffer and use a render buffer and make sure that everything that we want to get post-processed gets rendered to the FBO so bind the FBO before we render the scene to the screen. we also want the water to get render to this but we don’t really want the GUI to get rendered to this we don’t need them to be part of the post-processing so unbind the FBO before the GUI is rendered then we need to remember to clean up the FBO after the project has closed.

Post-processing effects are divided into two types: One is full-screen post-processing effects, another one is local post-processing.

Full screen post-processing

Full screen post-processing is applied to the entire screen, usually after the rest of the image has been rendered such as color filters, depth of field and full screen bloom. so next I will introduce a full screen post-processing effect: Motion Blur.

Motion Blur

Assuming there is a fast-moving pen in front of your eyes from left to right, you will notice that a big blur because you cannot see the pen clearly. The same phenomenon also take place with photo and video cameras. For example, photographed a person who is always swaying so that a blur figure in the photo because the human eye and cameras do not take in information at an infinite rate but take snapshots of what they see many times per second. If an object is moving faster than the rate at which images are captured, it will appear to streak across the image.

Knowing how the effect happens in real life can tell us to implement it in our render system. The ideal approach is to determine the movement speed of each pixel in the scene from one frame to another and use this information to blur the image correctly.

This approach is too complicated to implement at this point in this article. Another approach would be to ignore the speed of movement and take the pixels from the previous frame and blend them with the pixels of the current frame. This approach is a gross estimation because it does not account for actual speed but has been taken in many of today's video games and is the approach I will take in this article.

My method may seem like a rough approximation of the real effect, and you may think it would not run. A common misconception in computer graphics is that unless you do it the exact way it appears in real life, it cannot look good. When you read this article, you will learn that the essence of computer graphics is not how exactly you do something, but how convincing it is to the human eye. Human vision is a very subjective thing and does not perceive everything equally, just as you can't assume the intensity of a color is the simple average of all its components. Before you can write a motion blur shader[16] you should understand the process of how the effect happens using Render System. You will need to render your scene to a render target and then blend the current result with the previous output from the previous frame. You can see the whole motion blur process is simply a recursive loop, where the result from the previous frame is used to render the next frames. It is necessary to introduce some basic elements for motion blur. First, how to render a scene to a texture. Second, how to use the texture.

Rendering to a drawing board

Generally, it is simple to implement a screen effect[10]. First, you do not need to use the screen buffer, only render the scene to a texture. And then, you have many choices to filter this texture which stored the scene.

After filtering this texture, you can put it into the screen buffer. And then it will be appeared to the user. So, before implementing some of this kind of effects. You should know how to render a scene to a texture. It will be worth for other kinds of effects. Our developing software is RenderMonkey,

We will use a render target which is popular used in DirectX[17] and OpenGL[35]. It can be called by different rendering API, and to invoke the hardware to output the result to the texture you choose.

You can create the render target as a pass. In the same way, you also can create the second pass, the third pass and so on. Notice that the user cannot see anything because the whole scene is rendered to the render target, rather than to the frame buffer. At this time, you can assume everything goes well. The next step is that create a new pass to copy the result from this render target to the frame buffer. Finally, it will be presented on the screen.

So now, you can apply the texture to the object and projecting it onto the screen. The whole process is shown in figure 5.3

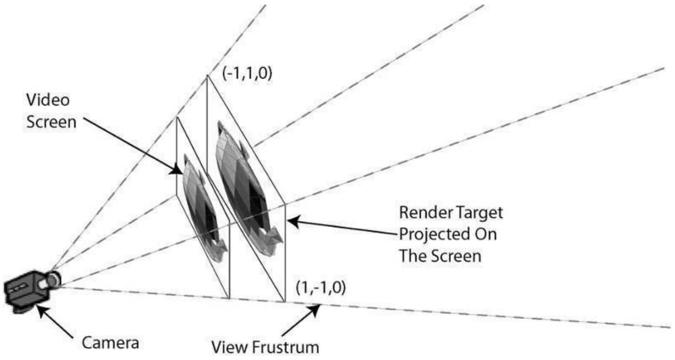


Figure 2 www.nvidia.com

Texture transformation

If using the whole texture, you will need to know that the texture coordinates are (0,0), (1,0), (1,1), (0,1) in 2d space.

You will consider one thing that the pixels on the screen and the texture are not completely match because the hardware just make the coordinates of screen correspond to the center of the texture pixel. So, you need to offset the texture slightly such as half a pixel to correct this problem which is some unexpected filtering.

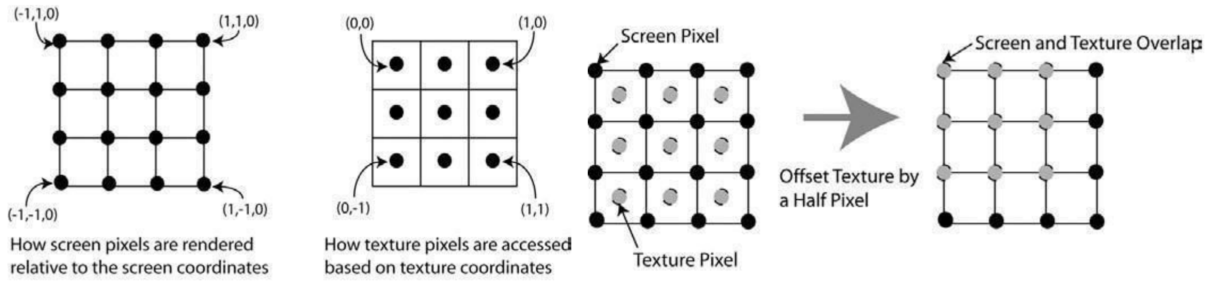

(-1,1,0)
(1,1,0)
(-1,-1,0)
(1,-1,0)
How screen pixels are rendered relative to the screen coordinates
(0,0)
(1,0)
(0,-1)
(1,1)
How texture pixels are accessed based on texture coordinates
Screen Pixel
Texture Pixel
Offset Texture by a Half Pixel
Screen and Texture Overlap

Figure 3 *www.nvidia.com*

Rendering the Render Target

It is time to create the pixel shader. The input parameter should be the texture coordination[21]. And a sampler texture. Finally, return the sampled color according to the coordinates.

```
sampler Texture0;

float4 ps_main(float2 texCoord: TEXCOORD0) : COLOR
{

// Simply read the temporary texture and send

// the color to the output without manipulating it

return tex2D(Texture0, texCoord);

}
```

You should have enough information to implement a motion blur effect so far.

The workflow process

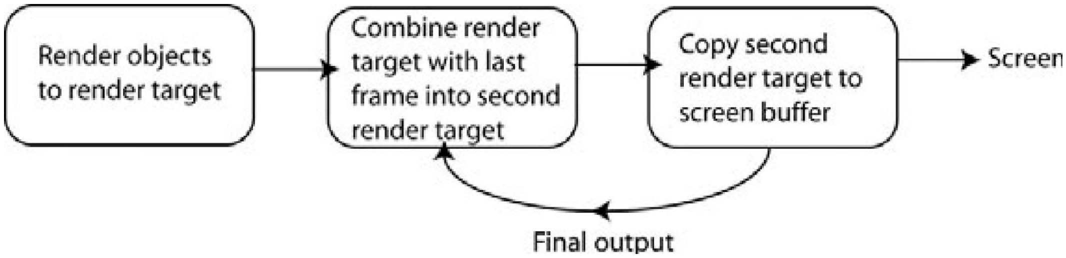

Render objects to render target
Combine render target with last frame into second render target
Copy second render target to screen buffer
Screen
Final output

Figure 5 www.nvidia.com

(The first render target was used by the second render target)

.

Creating the Motion Blur Shader

After you should understand the process of the effects using RenderMonkey. you can create a motion blur shader. As a matter of fact, the process is to render the whole scene to a RT and then blend the current result with the previous output from the previous frame.

Selecting a Template in RenderMonkey

First, confirm that you have the correct template for your shader. This template has been supported in RenderMonkey.

Add a new pass for blending the result

- Using the template shader developed in RenderMonkey.
- It already renders the whole scene to a render target and supports a pass to duplicate a render target to the screen.
- According to the diagram in Figure 1, the last thing is a pass that is responsible for blending the result from the previous frame with the current result. Therefore, you need to add another pass to your effect.

- If the new pass is added at the end of the effect, you will move up the new pass because the render system actually renders them sequentially in the order they are appeared within the List in the RenderMonkey

Adding a new render target

In addition, adding a new render target which is used to not only keep the previous frame's render result, but also track of this to your effect.

At this time, you have the RenderTarget which render the objects from the scene and blending pass, and then you can blend the two render targets together. Adding a variable named blur_control to the effect to control how the two textures will be combined is worthwhile because you can adjust this value to see the changs of effect immediately.

Blending in Pixel Shader

- Adding the RT2 and blur_control to the Pixel Shader
- Blend two texture together.
- Using the blur_control to adjust the desirable effects

Algorithm

Overview

The algorithm consists of the following two rendering passes:

Unblurred geometry

- The moving scene geometry is drawn, textured and shaded (simple Gourad interpolation).
- This is rendered directly onto a framebuffer object(FBO)

Blurred geometry:

- The static scene geometry is drawn, textured and shaded(simple Gourad interpolation)
- The skybox is drawn (using a cubemap texture) onto a screen-aligned quad at max depth distance.
- The "strtched"moving geometry is drawn(with depth test disabled). In this step, the FBO color attachment rendered earlier is used for supersampling.
- All of this is rendered directly onto the default framebuffer.

Formula

- Vertex Shader

  ```
  Out.texCoord.x = 0.5 * (1 + Pos.x - viewport_inv_width);
  Out.texCoord.y = 0.5 * (1 - Pos.y - viewport_inv_height);
  ```

  Since the reason of the hardware, the texture coordinate has to get a half offset to overlap the pixel on the screen correctly.

- Pixel Shader

  ```
  float4 col1 = tex2D(Texture0, texCoord);
  float4 col2 = tex2D(Texture1, texCoord);
  return lerp(col1,col2,blur_factor);
  ```

Using the weight blur_factor to control the blur extent. If the blur_factor is more bigger, then the blur extent is more bigger because it tend to the col2 which come from current frame. Or it will tend to the col2 which is the result of the previous frame, the blur extent is more smaller.

Analysis and summary

This article is a simple implementation of a filter for motion blur.It mainly take advantage of difference position between the previous frame and the current frame. Using a weight variable (blur_control) to control the blur extent in a linear function in pixel shader.

This method has some constraints:

- The effect act on some specific object which is fast-moving because it is not a post-process effect.

- The performance is dependent on the scene complexity. Such as there is only one car in the camera, the car consists of 5000 vertexes. GPU just process and blur the 5000 vertexes. That's ok. But if,

  there are 100 cars in the camera. 500000 vertexes will be blur by the GPU, that's bad design because the frame will be stuck, the value of the frame may be less than 10.

- This method is computed each object which is fast-moving. So, it's not dependent on camera motion

For example:

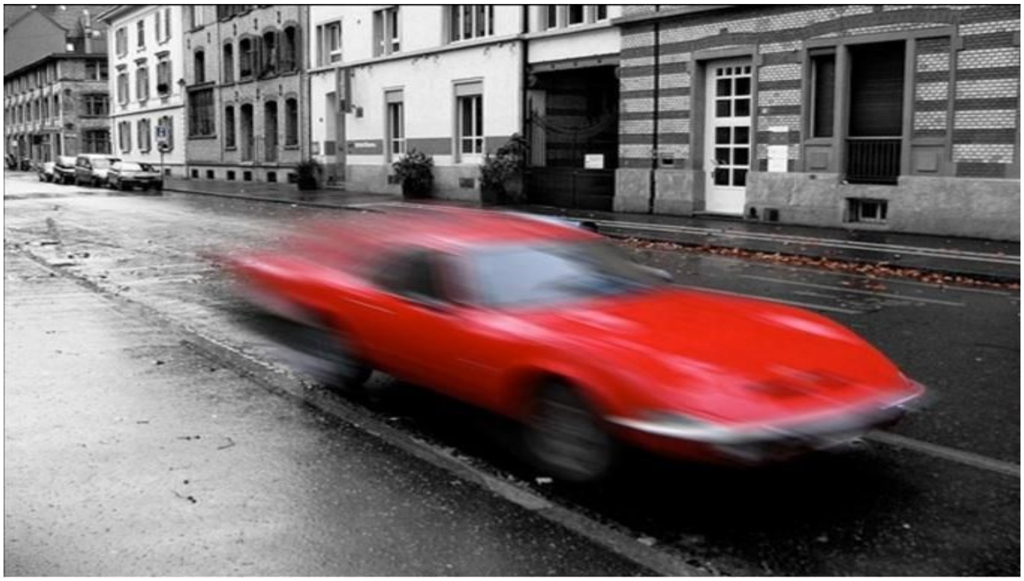


Figure 6 *www.nvidia.com*

In addition, there is another algorithm is a post-process that to be processed on a 2d texture. The performance is not dependent on specific objects.This method is an implementation of computing the pixel by vector direction.

- It is only computed in screen space so that the rendering system does not need to be changed for the effect
- The performance is not dependent on the objects in scene. But it will be affected by the screen resolution. The screen resolution is more bigger, the performance is more lower, or more higher.

- This method can be used the whole scene, and does not care how complex the scene is because it was processed only on a 2d texture.
- If the camera is attached an object no matter the object is moving or not, as long as the object is in the range of camera, it will be blurred.

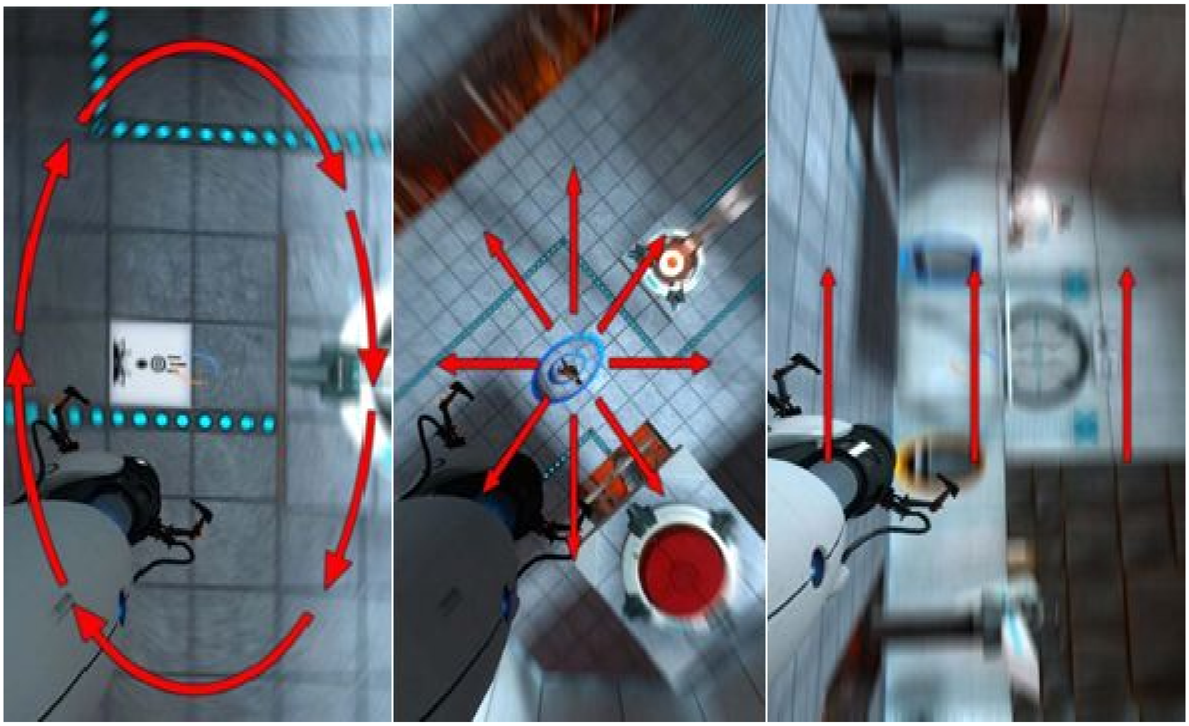


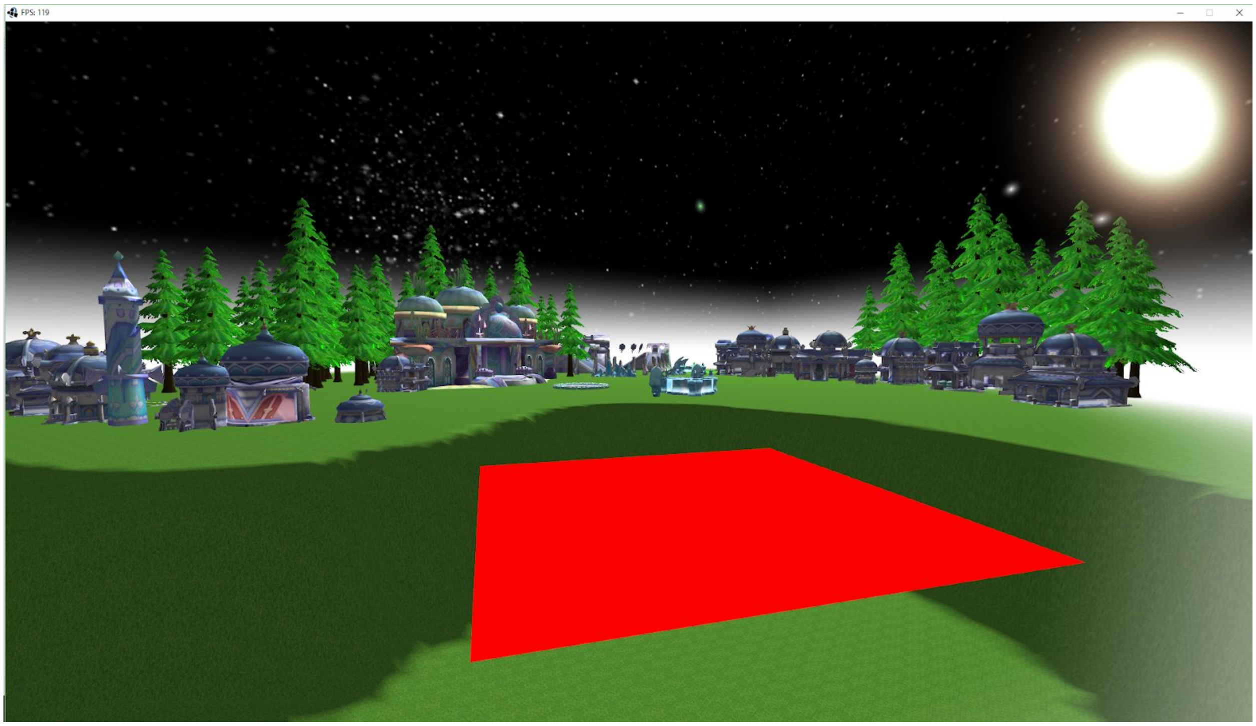
Figure 7 *www.nvidia.com*

Local Post-processing Effects

Water Rendering

This demo will go through the process of creating some rippling, reflective, refractive sparkling water that is all post-processing effects, so that's the programmable pipeline and not the old fixed function pipeline. Here is aimed at people who already have a good understanding of the basic concepts of the modern rendering pipeline, and who understand the basics of GLSL, the shading language. And to demonstrate the water using the 3D engine that has been creating in the project.

So basically everything that needs to render the scene from scratch. If you don't want to put all of this into one method that's fine, but whenever I call this method just remember that it means that you have to carry out the full rendering process, because for rendering water we will actually need to do this multiple times every frame. In the scene, and all of the GUI renderings is still done separately. So I am just going to give a quick overview now of how to be achieving the water effect. The water is actually just going to be a completely flat quad and using some clever shader techniques to make it look like it's a rippling reflective surface.


FPS: 119

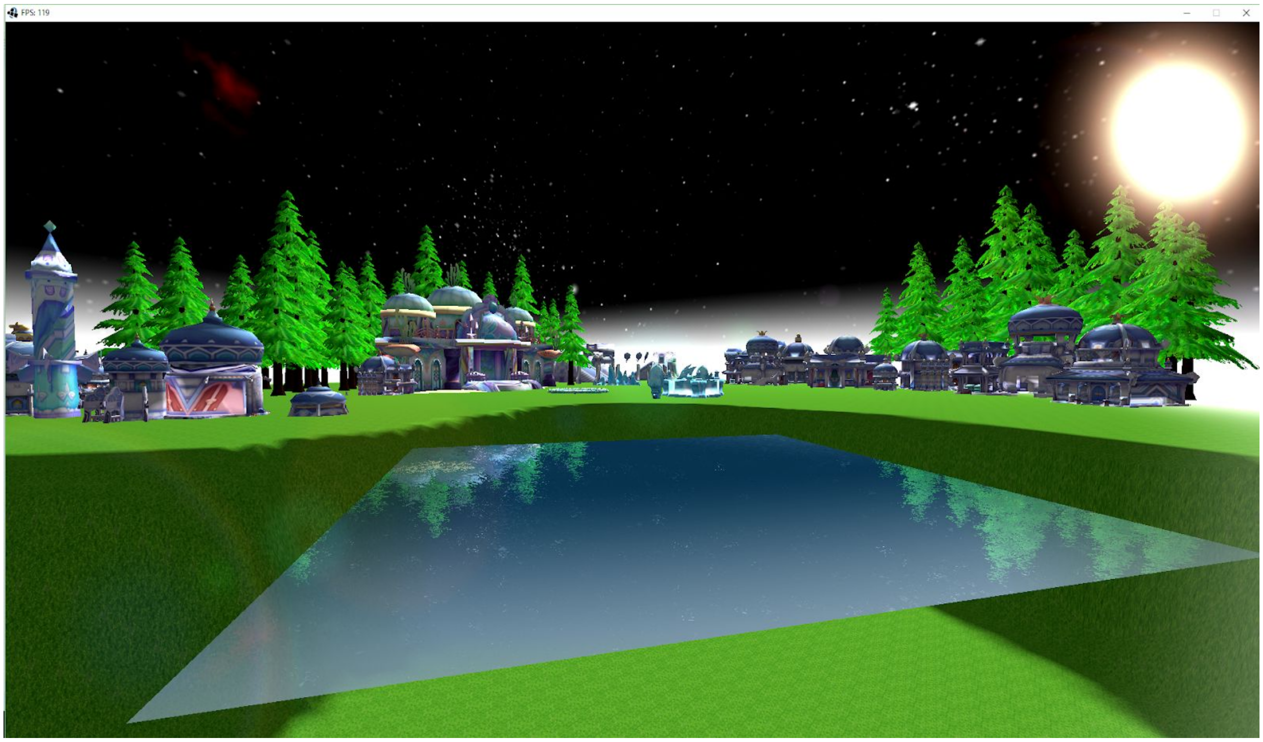

FPS: 119

Figure 8

The first thing that is to render the scene to some textures. This is basically like taking a picture of a scene and storing the result in a texture that can then use just like any other texture. Before rendering the water, each frame we will take two pictures of the scene, one of everything under the water which will be saved in the refraction texture, and one from a slightly different camera angle of everything above the water which will be the reflection texture. To use these two textures to texture the quad when rendering the water quad, but instead of just texturing the water normally it will first need to distort these two textures using something called a DUDV map. The distortion will change over time and give the impression that the textures are rippling. Then blending these two distorted textures together on the water's surface using the Fresnel effect, this just determines how much of each texture should be seen based on the viewing angle, so that when looking straight down into the water it appears a few transparent, but when looking across the water at a low angle it looks more reflective. Then adding some lighting calculations with the help of a normal map to make the water look like it isn't flat and adding some specular lighting to make the water glisten in the sunlight. Finally, it will be softened the edges of the water and possibly make the deeper areas look a little bit murkier using a depth texture which will be created when we render the refraction texture.

First, it will be rendering to that is called Frame Buffer Objects so usually when the scene is rendered, the result is shown on the screen but another way is to render the scene and store the result in the texture instead of putting the results onto the screen once the scene is rendered to a texture and then to use that texture just like any other texture and render it onto 3D objects or whatever you want so that's what is shown on the screen now what I have done here is to first render the scene to a texture and then render the scene as normal to the screen but the texture that is created above is used to texture this GUI in the top left corner here and because rendering a scene to that texture every single frame the GUI texture is constantly being updated so what has all this got to do with rendering water well if you have a look at this water here you can see the reflection of the scene on the surface of the water and the way that I have done that is exactly the same thing that I did with the GUI here. Firstly, render the scene to a texture and then render the scene to the screen as normal but used that texture that is rendered earlier to texture the water's surface and if choosing the correct camera position when doing that first render pass then it will create an accurate reflection of the scene so that needs to use something called Frame Buffer Objects.

When rendering a scene, object and terrain one by one but they don't appear on the screen until update the display. when rendering an object, it first gets rendered to the frame buffer. The frame buffer stores all of the data about what's going to be rendered onto the display it has a color buffer which is basically a 2D array of pixel colors and it's this that gets shown on the screen when we update the display as we render each object the scene gets created in the color buffer there's also another buffer called the depth buffer which stores the depth information of each pixel that's in the color buffer. when rendering an object with depth testing enabled the object will first be checked against the depth information in the depth buffer this tests whether the object your rendering should be in front or behind the other object that has been already rendered and it then discards any pixels that are hidden so if an object is rendered that is completely hidden by another object then the depth test will notice that and the color buffer won't be updated at all. Hence the object can't be seen on the screen there are also some other buffers like the stencil buffer but there is no need to worried about them since once rendering the scene has been finished to the frame buffer and update the display and the image in the color buffer is put onto the screen you can then clear all of the buffers and render the scene all over again for the next frame, however, we don't just have to render to this one frame buffer.

A customer frame buffer objects can also be created once a frame buffer object is created and it can be given attachments such as a color buffer and a depth buffer and then these objects are rendered to this frame buffer instead of the default frame buffer so before rendering anything there is a choice where the object to be rendered to the default frame buffer which would eventually be displayed on our screen or to render to one of the customer FBO behind the scenes frame buffers so here choosing to render to our frame buffer object and the depth buffer and the color buffer of this frame buffer get updated accordingly but the default frame buffer is still completely empty then switch back to the default frame buffer render some trees and houses, for example, update the display and then only the default frame buffers color buffer is shown on the screen these other frame buffer objects are off-screen buffers and their contents don't affect what's shown on the screen when we update the display so this frame buffer object seems a bit pointless right now but one nice thing that is to use textures as attachments here so instead of this color buffer just being a load of data in memory somewhere it's now an actual 2D texture just like any of the other textures in the scene this means that when rendering this frame buffer object, the color buffer gets updated.

Updating this 2D texture so to render a house to this frame buffer which would update the texture here and then switch back to the default Frame Buffer render a picture frame or something and then to use this texture here to texture the picture frame just like texturing any other objects in the sense so a texture can be created using Frame Buffer objects. so for the purpose of rendering water, two frame buffer objects both with depth buffers and color buffer attachments are going to be creating in one of these will render the reflection texture as mentioned earlier and in the other one to render the refraction texture and use these textures in future both of the color buffer attachments here are going to be textures of course so that the water's surface could be textured by using them. but the depth buffer of the refraction texture is also going to be a texture because this will basically be storing the depth of the water and sample that in the fragment shader in the future when rendering the water so that we can do some nice depth effects so in this case, it would just be easier.

Design the planes

Clipping planes is to specify a 3d plane in the world and then all the geometry which is outside of that plane is simply not rendered so as you can see here to specify a horizontal plane at a certain height and everything to one side of that plane is rendered but everything to the other side is not, this is going to be very useful. When rendering the refraction and reflection textures for the water when rendering the refraction texture that only want to render geometry which is under the water's surface because stuff that's above the water's surface isn't going to be refracted and when rendering the reflection texture that only renders geometry which is above the water surface because geometry under the water would obviously not be reflected on the water's surface. It's important that using clip planes for two reasons firstly it will help a lot with performance because all of the clipped geometry doesn't have to be processed by the fragment shader and that will save a lot of time especially when rendering the refraction texture because everything above the water won't be processed also if don't use a clip plane when rendering the refraction texture then that would end up with this kind of problem here where objects above the water are also refracted on the water's surface and that's obviously something so that using an inbuilt variable in the vertex shader called GLClipDistance is the best choice.

GLClipDistance is an array of floats where each element in the array corresponds to one clipping plane because using one clipping plane at a time so we will always be using GLClipDistance 0 and the first thing that before using this variable is to enable it so let's enable GLClipDistance 0 and if using other clip planes that would have to enable them as well. So this GLClipDistance variable allows to tell OpenGL how far each vertex is from the clipping plane OpenGL doesn't care about the clipping plane at all it doesn't want to know which clipping plane it is. It doesn't care about the clipping plane is OpenGL just wants to know the signs distance of each vertex from the plane whichever plane wherever that plane may be. If this distance is positive then it means that the vertex is on the inside of the plane the correct side of the plane and so won't get culled. If the distance is negative then it means that the vertex is on the outside of the plane on the culling side of the plane so giving the distance for each vertex and then OpenGL interpolates these distances over the whole geometry and then any part of the geometry with the negative distance is culled and isn't rendered and anything with a distance of zero and over is rendered so let's text that out here in the vertex shader and all is to set the GLClipDistance as minus one for every single vertex this tells OpenGL that every single vertex is outside the clipping plane and should, therefore, be culled and so if running this now that none of the entities are being rendered because they are all being culled alternatively that could set this

GLClipDistance to a positive number and that would tell OpenGL that all of the vertexes are inside the plane but that’s obviously not very useful it's necessary to define a plane and then calculate the distance of each vertex from that plane so that telling OpenGL to do this.

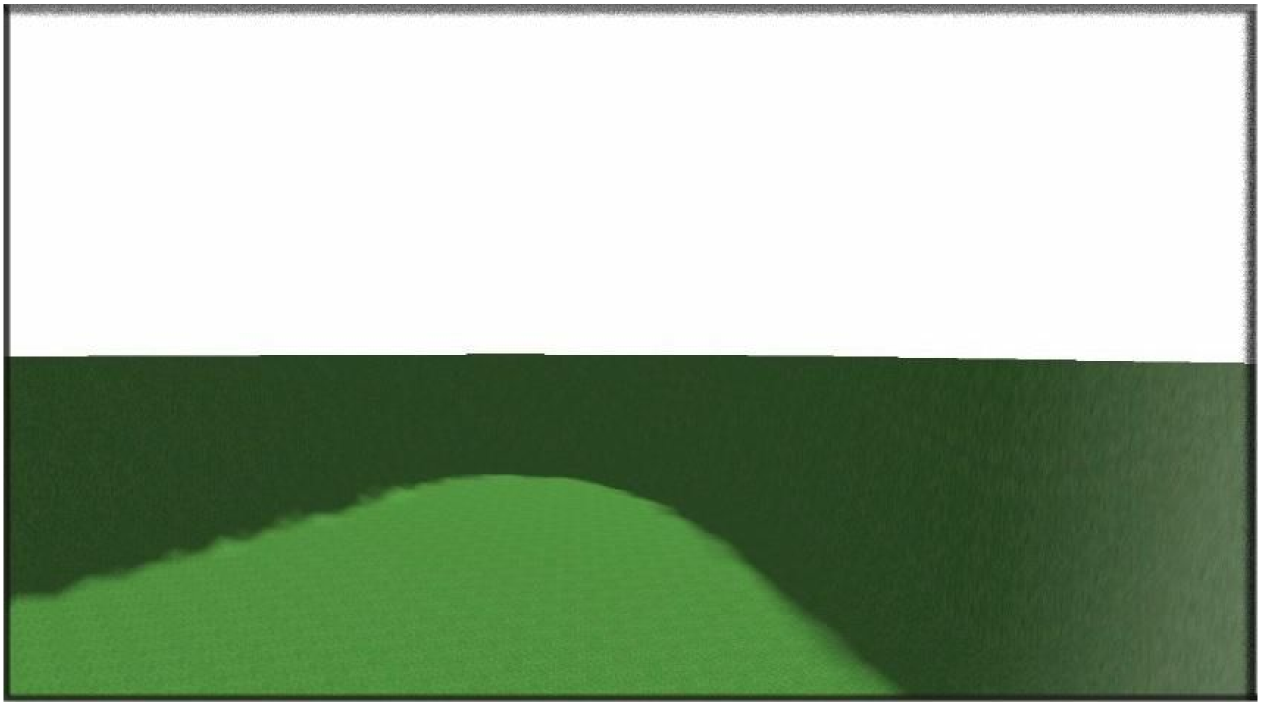
The first thing is to define a plane and to define a plane needs to know the equation of the plane now to explain plane equations won't be shown here. Now, just using horizontal planes anyway which makes things very easy because the normal of the plane is always going to be zero one zero pointing straight upwards or zero minus one zero pointing straight downwards. if letting everything above the plane to be culled and then the distance D from the origin is just going to be the height of the plane so to define a plane these values are needed and to store them in a 4d vector so now define a plane here in the vertex shader and create a horizontal plane which curls everything above a height of 1 or to choose a different height depending on the height of the entities in the world. So now that a plane is already here, and then to find the distance of each vertex from that plane to do that it has to take the dot product of the position of the vertex in the world and the 4d vector created using those values from the planes equation and this will return the distance of the vertex from the plane then tell OpenGL using the GLClipdistance 0 variable so this should now work and if running this, it has indeed curled everything above the height of 1 so now to specify the plane in the java code instead of having to hard-coded into the shaders so now to find the plane here that is to make this a uniform variable so that values can be loaded up to it from the java code so now doing the usual thing in the static shader that will do every time when adding a new uniform variable so it needs a new integer to hold the location of that uniform variable then it need to get the location of that uniform in the get all uniform locations method and don’t forget to spell plane correctly and now a method is needed that can load up a

4d vector to that uniform variable so this will be called LoadClipPlane and it will take in a 4D vector and have to go into the shader program class quickly to add a method here that can load up a 4D vector so to copy the one that loads up a 3D vector and then just change that from vector 3F to vector 4F from GL_uniform3f to GL_uniform4f and then add in that fourth vector.

Mapping the projective texture

Applying the reflection and refraction textures to the water surface using projective texturing and starting off in the water fragment shader here so to sample two 2D textures is needed here. One for the reflection texture and one for the refraction texture even and these are both going to be uniform variables so now sample them in the usual way so to get the reflection color that's just going to be using the texture function to sample the reflection texture and sample it at the texture coordinates that taking into the fragment shader and to do exactly the same for the refraction texture here so to sample the refraction texture this time and once again to sample it at those texture coordinates and now that getting these two colors and simply to mix them equally together so using a mix value of 0.5 so now that need to move into the water shader class and do the usual stuff so that need an integer for both the reflection texture and the refraction texture uniform variables so create two integer and then of course to get the location of both of them so location reflection texture equals get a uniform location and then make sure to spell reflection texture correctly and then do exactly the same for the refraction texture and make sure that you have got location refraction texture equals get uniform location refraction texture now a method is need to be created that's going to load up an integer to each of these sampler 2D textures and the multi texturing is very useful technology to indicate which texture units that should be something from so for the reflections extra to be sampling from texture unit 0 so to bind our reflection texture to texture unit 0 and then for the refraction texture to be sampling from texture unit 1 so make sure that second one is location refraction texture and it’s something from texture unit 1 so into the water renderer cast now this is going to need to have access to those reflection and refraction textures so it’s going to have to take in the water frame buffers and have any instance variable for it up here and then set it in the constructor and also just after starting the shader here it will call that connect texture units method to connect up those texture units so now if going down a bit in the prepare render method that needs to bind the correct textures to the correct texture units so to activate the texture unit before we can bind anything to it calling GL active texture and then choose which texture unit to activate and act make texture unit 0 the active texture unit and then to bind the reflection texture for that texture unit as mentioned earlier so just call GL_bind_texture that is a 2d texture and to find the reflection texture to texture unit 0 and then, of course, to do exactly the same for the refraction texture but binding this to texture unit 1 and don’t forget to change this to FBOs get refraction texture so to get all that set up so the water frame buffers need to be created a bit earlier now before the water renderer and any of this GUI stuff won't be needed anymore so to get rid of that and put the water frame buffers object into the water renderer constructor and that should be it so running that both the refraction and reflection textures are textured onto the water quad now but it looks horrible it’s a total disaster and that’s because we need to use projective texturing so the easiest way to show how projective texturing works is to actually do it in an image editor so here is the reflection texture for the scene at a certain point in time and to make it fill up the whole display here before doing anything else though flipping it upside down because it is a reflection and then onto this to overlay a picture of the whole scene which would be rendered to the screen with the water quad as well and this was taken at the same point in time as the reflection texture so to texture the reflection texture onto this blue water quad and I can actually do this here by simply rubbing away this layer to reveal the reflection texture below and the reflection texture is now perfectly textured to the water quad. Creating a nice reflection of the scene in the water and the same of course would work with the refraction texture.

Reflection texture



Refraction texture

To sample the texture for this point on the water quad that need to find out the correct UV texture coordinates to sample this texture and if finding out the screen space coordinates for this point on the water quad then those exact same coordinates to sample the reflection or the refraction texture can be used to so how to find the screen space coordinates of a point on the water quad well. Next, converting two different spaces is going to be shown in the water vertex shader. Now the clip space coordinates has been created because the vertex position has been multiplied by the model matrix, the view matrix, and the projection matrix so the next step is to convert to normalized to face space and it needs to do perspective division that needs to divide the clip space coordinates so in the water fragment shader that will be no longer needed to take in the vector texture coordinates so just delete that and in the water vertex shader. The vector texture coordinates will no longer be outputted so that will be deleted and the texture coordinates down here won't be needed to calculate that however is the clip space coordinates of this vertex and that’s going to be a vector for called clip space and then as mentioned earlier the clip space coordinates are equal to the vertex position multiplied by the model matrix view matrix and projection matrices so all those matrices won't be doing the calculations twice just set glposition equal to clip space so now in the water fragment shader that vector will be taken in for clip space so in effect for clip space and that clip space will be needed to convert to normalize device space and as shown earlier that can be done by using Perspective division and to do that first it's not only the X and Y components here but just divide by the W component to carry out to the perspective division and to convert to normalize device space so now that can get the coordinates of any given point on the water quad in the coordinate system. But to sample a texture, the coordinates in this coordinate system has to be given. Luckily, the conversion between the two coordinate systems is very easy and The first thing is to divide by two and then add 0.5 and that will get the screen space coordinates of any given point of the water quad in this coordinate system and then we can use those exact same coordinates to sample the textures so to divide by 2 and add 0.5 and that will now get the screen space coordinates so the refraction texture coordinates simply going to be equal to the normalized device coordinates that have just calculated this lines a bit pointless but just to make things obvious as to what this part is doing and then the reflection texture coordinates are also going to be equal to those normalized device's coordinates those green space coordinates that calculated but because it’s a reflection so that need to invert the y coordinate there and tip it upside down so that it’s reflected so now to use the refraction texture coordination to sample the refraction texture and use the reflection texture coordinates to sampled the reflection texture to make sure that is the right way round there and that should be everything so that should now have implemented projective texturing and it has and it is all working fine the reflection and the water looks slightly transparent as well so that is it for here. Next, making the water look like it's rippling and doing that by using DUDV maps.

Using the DUDV Maps

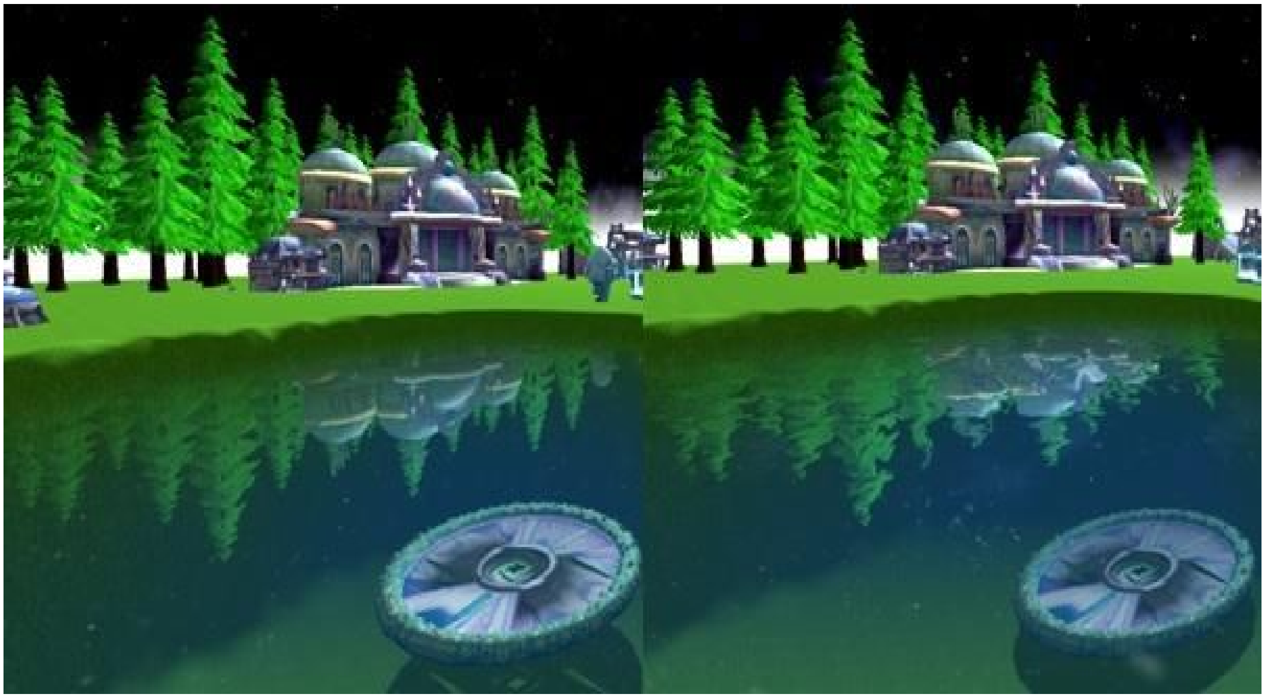


Figure 9

Now making the water look like it's rippling by using something called a DUDV map so the way that making the water all rippling is by distorting the reflection and refraction textures and to distort a texture. An offset can be added to the texture coordinates that are used to sample it so, for example, the reflection texture can be distorted by adding an offset to the reflection texture coordinates and when running this it has shifted the reflection to the right. The same thing also could be done and shift the reflection to the left like this but this just adds a constant distortion over the whole water quad to make this look realistic the distortion needs to be different at different points on the water's surface so this distortion value will be needed here to be different for different places on the quad and this is where the DUDV map comes a DUDV map is just a simple texture with a lot of red and green wobbles on it so a DUDV map is going to represent the distortion at different points on the water's surface the texture is obviously full of lots of red and green colors and these red and green values are all 2D vectors which can be used as the offsets to add to the reflection and refraction texture coordinates so to map the DUDV texture to the water quad and then we will sample it for every point on the water surface to get the distortion for that point however the red and green values are always going to be positive in the DUDV texture. It is not allowed to have a negative color so the distortions would always be positive but now a better way is to be able to have both positive and negative distortions for this to look realistic. So to convert these red and green values is needed before using them this is going to be very easy to do currently the color values are all between zero and one so to convert them to be between minus 1 and 1 and simply multiply by 2 and to substract 1 and this conversion will be done in the fragment shader in a minute so to start off in the water fragment shader and because the DUDV map is just texture and it also need a sampler2D so that it can be sampled and get those distortion values in the water shader class so creating an integer which is going to hold the location of that uniform variable. Getting the location of that uniform variable and filling DUDV map correctly and then to connect the texture unit to indicate that sampling the DUDV map from texture unit 2 and to remember that that's where the DUDV map need to be bound so the DUDV map needs to be loaded up so to create a constant string up which is going to be the name of the DUDV map and that is called water DUDV and then an integer is needed which is going to hold the ID of the DUDV map texture then load it up in the water renderer constructor using the loader texture method so that loads up DUDV map and then in the prepare render method it needs to be bound and as mentioned just a second ago it needs to be bound to texture unit 2. so now to go into the water vertex shader and mess up a little bit because the output texture coordinates is needed in that calculation of the vertex shader and these textures' coordinates are going to be tile with the terrain textures so a tiling value is going to be set and then the texture coordinates are going to be multiplied by that two tiles that two tiles the texture the DUDV map so now in the water fragment shader taking in those texture coordinates which are just simple texture coordinates for that quad and that DUDV map is going to be sampled to get the distortion which is called distortion. One, for now, is to sample the DUDV map and sample it at those texture coordinates but there may be one question why does it only care about the red and green values because that’s where the distortion is stored and that conversion has to be done as mentioned in the explanation earlier so now getting the distortion and use this to distort the refraction and reflection texture coordinates and to do that simply by adding the distortion to that texture coordinates so doing that for the refraction and for the reflection texture coordinates and then if running that something should have happened there are a lot of distortion now but it’s perhaps a bit too much distortion so back in the fragment shader, changing the strength of the distortion and to create a constant here which is going to indicate how strong that distortion should be and I am going to set that to 0.01 but it can be any value. and to multiply it by the wave strength which should hopefully make the distortion a little bit less and if running that has indeed worked and that the distortion is now a lot more pleasant however there is now another problem and that is the wobbly glitch right down at the bottom of the screen there because of projective texturing the reflection and refraction texture coordinates at the bottom of the screen are going to have a y-value of almost zero and because the distortion adds an offset to the texture coordinates they are sometimes going to go below this causes the textures to go off the bottom of the texture and then wrap back round to the top of the texture because this causes that nasty glitch so to fix this problem to clamp the texture coordinates to make sure that they never to too high or too low so using the clamp function to clamp the refraction texture coordination and clamp them between zero point double zero one and zero point double nine that will be the same for the reflection texture coordinates but each component has to be done separately here so first for the x component it can be clamp so the x component is clamped between again zero point zero one and zero point nine nine nine but for the y coordinate this is flipped for the texture coordinates in the last episode so the clapping is also flipped so we clamp between minus zero point nine nine nine and minus zero point double zero one so that should now have clamped the texture coordinates so if running that that problem has no longer appeared at the bottom of the screen, however, the water still looks very static. the distortion is completely static it's not moving at all and that's very unrealistic because in real water the ripples would be moving over the surface so these ripples needs to be moved so the way that to get the water to look like it's moving is to have an offset for where we sample the DUDV map and that offset is going to be called move factor and changing this offset over time which will make the water look like it's moving so in the water shader. The move factor uniform variable would be created so to get the location of it and a method is needed which will be able to load up the float to that move factor uniform variable to change the offset over time which will make the water look like the distortion is moving so in the water renderer this move factor needs to be changed and move it at a certain speed and that speed can be defined to the wave speed as 0.03 or any else and then to create the move factor variable here which would be initialized as zero then in the prepare render methods every single frame to increase the move factor by the speed by the wave speed and the speed is a distance per second so to multiply that by the number of second that have passed and then we want to make the move factor loop back to zero when it gets to one which has done by doing that and then to load up the move factor to the shader by doing shader mode move factor and then in the fragment shader now this move factor value has been changing over time so to add an offset to the X texture coordinate for where to sample the DUDV map and that will make the distortion move in the X direction over time and if you have a look at the trees in the reflection there you can see that the distortion is indeed moving but the distortion itself isn't actually changing it's just moving over the surface so making this even more realistic by actually sampling the DUDV map again and having another distortion move in a different direction so to sample the texture again the DUDV map again this will be distortion too and for this it will be just changed a few things flipping the x texture coordinate by making it negative and making this distortion move in the Y direction as well just to be a bit different from the first one and now adding those two distortions together to create that final total distortion which will look a lot more realistic so total distortion equals distortion one plus two distortion two and then to use that here so running that and hopefully there are some pretty realistic looking distortion and that is now the case so it's looking pretty nice now the distortion is changing nicely over time and to mess around with those variables but one thing that is to add a bit of a blue tint to the color of the water so to add the color at the end of the water fragment shader and to mix that final color that has been calculated and I am going to mix it with a bluish color of bluey green so using the mix function and mix it with this blue color and just to add 0.2 of that blue color so not very much just a very slight tinge to the water and it looks a little bit more blue so that is going to be it.

Taking advantage the Fresnel effect

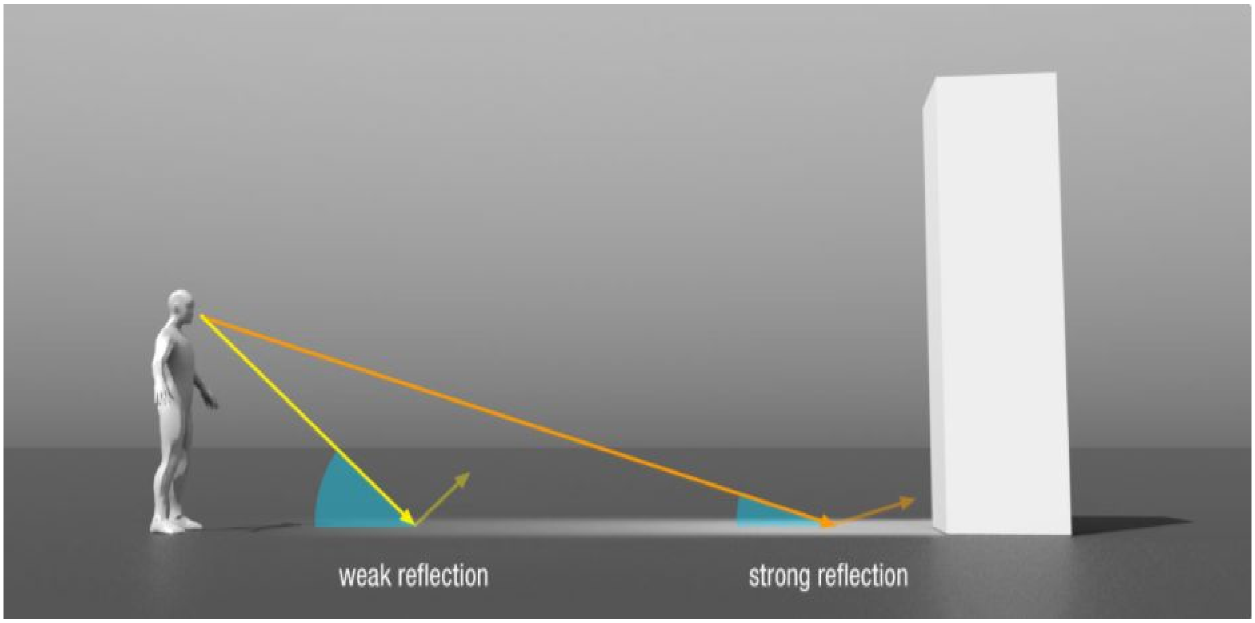


Figure 10

Adding the fresnel effect so the final effect describes how the reflectivity of the water changes depending on the viewing angle so water appears more transparent when looks out from above but more reflective when looked at from a low angle like this a simple Fornell effect is actually pretty easy to calculate so here we have got a water surface with the surface normal pointing upwards and we need to calculate how reflective the surface should be depending on the camera's position as I just showed when the camera views the water from above the water should appear more transparent and when viewed from a low angle it should appear more reflective if I add in the vector pointing from the water to the camera you can see that the water should appear more transparent when these two vectors are pointing in the same direction and more reflective when the vectors are pointing in different directions so the more these vectors are pointing in the same direction the more transparent the water should be and if you remember from the lighting I mentioned above we can take the dot product of two vectors to find out how much they are pointing in the same direction the dot product returns one when two unit vectors are pointing in the exact same direction and it returns zero when they are perpendicular to each other so in this case the dot product of the waters normal and the vector pointing to the camera represents how transparent the water should be and that is easy to show this effect in real life here such as a bowl of water and if having a look at it from above the water is very transparent you can’t see the reflection of anything but if having a look at this from a low angle the water surface becomes very reflective so the first thing in the water vertex shader is to create a new uniform variable that can take in the camera’s position because the camera position is needed to be able to calculate the vector pointing from the water to the camera.

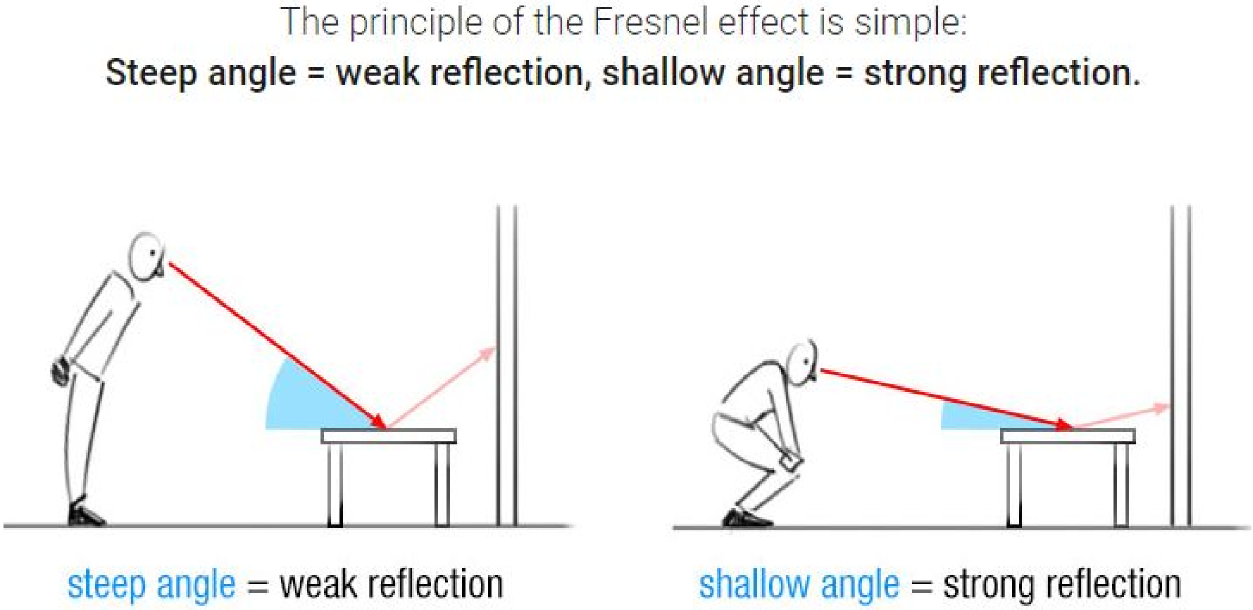

The principle of the Fresnel effect is simple:
Steep angle = weak reflection, shallow angle = strong reflection.
steep angle = weak reflection
shallow angle = strong reflection

Figure 11

In the water shader class to get the location of that new uniform variable and make sure to spell position correctly here then down in the load view matrix method to actually load up the camera position that's uniform variable which can be done by calling the load vector method so now go back into the water vertex shader and here to output the 3D vector which is pointing from the vertex to the camera and to calculate the world position of the vertex and that can be got by multiplying the model matrix with the vertex position so to copy that up here and then just put in the world position down here so that is unnecessary to do that calculation twice and now to calculate the two camera vector which is simply going to be the camera position minus the world position which is a 4 vector so that need to make it 3D by taking just the x y & z components now to go to the fragment shader the water fragment shader and this needs to take in that 3D vector pointing to the camera which will have been interpolated all over the court by now and the first thing that needs to do is to normalize it because the dot product needs the vectors to be unit vectors so normalize that camera vector and then to do that dot product calculation to find out how transparent the water should be so taking the dot product of the vector pointing to the camera and the water's surface normal which assumes is pointing straight upwards for now so 0 1 0 so now to mix the reflection and refraction colors together and just mix them half and half but now that refractive factor can be used to choose how transparent how much of the refraction texture appears and more of the refraction texture when looking down at the water and more of the reflection texture appears. when looking at it from a low angle it can change how reflective the water is and then raise the refractive factor to the power of a number and the higher this number the more reflective the water is so to choose a value 10 here which makes the water very reflective and it's only transparent when really looking down straight from above and in the same way to make the water less reflective if making that a smaller value like 0.5 and now the water is a lot less reflective than it was before so that is it.

Normal maps

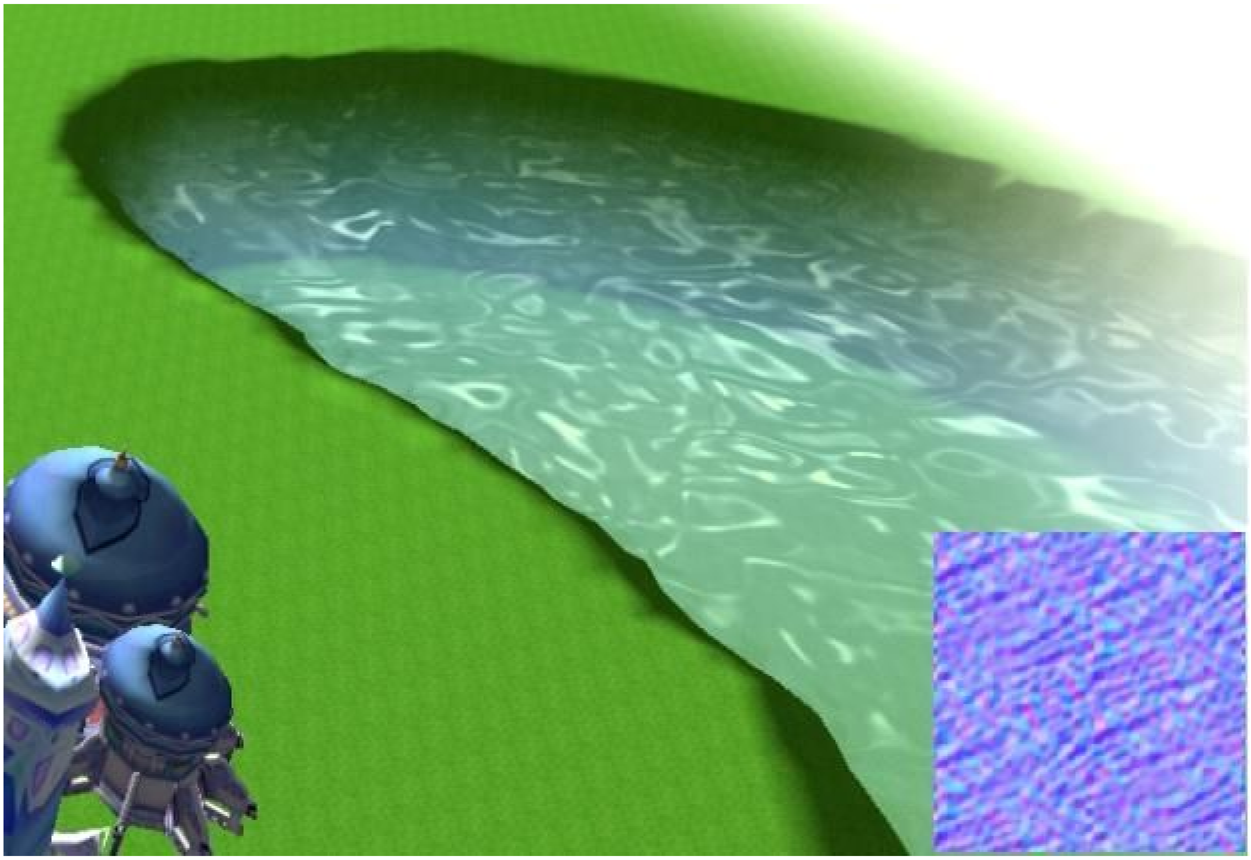


Figure 12

Adding specular highlights to the water. the lighting on an object is based on its surface normals unfortunately this water is a completely flat quad so the normals would all be pointing straight upwards and the lighting would be very boring indeed if this water model was more realistic it would have normals that look something like this so that is just pretend that this water model is realistic and to use normals like this even though this water is actually completely flat by using these normals that are all over the place like this it will make the lighting act as if the water wasn’t completely flat and as if it was actually bumpy and realistic but it still needs to generate these normals from somewhere and that’s where the normal map comes in the same way that the DUDV map can be used to indicate the distortion on different parts of the water. A normal map can be used to indicate the normal and different points on the water’s surface the pixel color at any point on the normal map here can indicate the 3D normal vector of the water at that point the normal map is mostly a blue color here because the blue value represents the up axis and in our case that’s y-axis so that will use the blue component of the normal map color to be the Y component in the normal vector the red and green components of the normal map can then be used as the x and z components of the vector but this is one problem because it is impossible to have a negative color. the components of the normal vector are also always going to be positive that’s not a problem for the y component because this surface normal should always to be pointing upwards to some extent but it isn't necessarily for the normal to always be pointing in the positive x and z directions to be realistic the normal should also be able to point to the negative x and z direction as well so just to use a quick conversion that has been used so many times to convert the x and z components to be between minus 1 and 1 instead of just 0 and 1 and that is extracting the normal vectors from the normal map so to jump into the water fragment shader code and the first thing is to calculate the distortion because there is a better way of doing it now so that will only need to sample the normal map once instead of twice just like sampling the DUDV map at two different places and then mixing the results together to get the final distortion value and that would have had to do the same with the normal map to get a similar effect with the lighting but this new method samples the DUDV map wants to get a distortion value and it then uses that distortion value as distorted texture coordinates which can then be used to sample the DUDV map again to get an actual distortion value but can then also be used to sample the normal map once to get the normal it also produces a slightly different look to the ripples which it looks a little bit nicer and switch to this method for the water as well back in the water fragment shader. To create a new sampler2D for the normal map because obviously it wants to sample the normal map at some point and then in the water shader class so to create a new integer which will hold the location of that normal map uniform variable then to get the location of that uniform variable by calling the gets uniform location method and putting in normal map and then to indicate which texture units the normal map should be sampled from and to indicate that it should be sampled from texture unit 3 because that’s just the next available texture unit in the water renderer.

To create a string constant which will indicate the texture file name for the normal map and it is called normal map then to need an integer to hold the ID of the normal map texture once it’s been loaded and then in the constructor. To actually load up the normal map by calling loader load texture and putting in the file name of the normal map then down in the prepare renderer in the prepare render method. To bind the texture to texture unit 3 so don’t forget to put text unit 3 there and bind the normal map to texture unit 3 so that can now sample the normal map from the fragment shader and so now to sample the normal map and get the normal map color, it will use the texture function to sample the normal map and to sample it at these distorted texture's coordinates the same coordinates that can be used to sample the DUDV map for now to set the output color of this fragment to that normal map color just to generate an idea of what the normal map looks like all distorted over the water’s surface and so that’s pretty much a representation of the normal of our water’s surface. So, go back and now to extract the normal from that normal map color in the way as mentioned earlier so for the X component that is just going to use the red value of the color and convert that so it’s between minus 1 and 1 for the Y component that is going to use the blue value and for the Z component. To use the green value which is also going to be converted between minus 1 and 1 and then we are just going to normalize that normal to make sure that it’s a unit vector so now there is the surface normal of the water but to calculate the lighting so a new uniform vector 3 is needed which is going to indicate what the light color is and in the water vertex shader a uniform vector 3 is also needed which is going to be the light position so that giving the shaders a bit more information about where the sun is in the scene so now in the water shader obviously the usual thing has to be done two more times so a location and integer are needed for the location of the light color and the light position uniform variables then to get the location of both of these making sure to spell the uniform names correctly especially for the light color one there make sure you have used either the UK or the American spelling whichever one to be used in the shader code and do the same for the light position and then to create a method which can actually load up this information to the shader so it’s going to take in a light which is going to basically be the Sun in the scene and then to load up the position and the color of the Sun so two location light color to load up the Sun dot get color and then for the position to load up the sun’s position to location light position so that will load up the information of the sun to these shaders then in the water renderer now a light needs to be taken in so that to send it to the shader so after starting the shader you can load up the light and then of course in the render method that’s also going to need to take in the sun which can then be put into the prepare render method and then in the main loop to need to put in the Sun or whichever light to use into the water renderer render method so going back to the water vertex to calculate a vector pointing from the light to the water and this is very similar to the sort of stuff in the lighting. To do the world position of the vertex the light position and that will get a vector pointing from the light to the water and then in the fragment shader in the water fragment shader now an integer vec3 is needed which is going to be that from light vector and there are a couple of other things that need to calculate the lighting a shine damper value is needed and a reflectivity value exactly the same as mentioned in that lighting knowledge above so now the vector is got which is the vector pointing from the light to the water and there is the view vector which is the vector pointing from the water to the camera and all the things are reflecting the light off the water using the normal vector and then to take the dot product of these two vectors to see how similar they are and the closer they are together the lighter is going directly into the camera and therefore the brighter the specular highlights appears so the specular highlights have been calculated that can be simply added on to the color of the water and it is necessary to convert it to a vector for first and the alpha component can be set to be zero because adding any extra alpha is unnecessary there and then if running that the water should now have some very nice specular highlights which depend on the direction of the light.

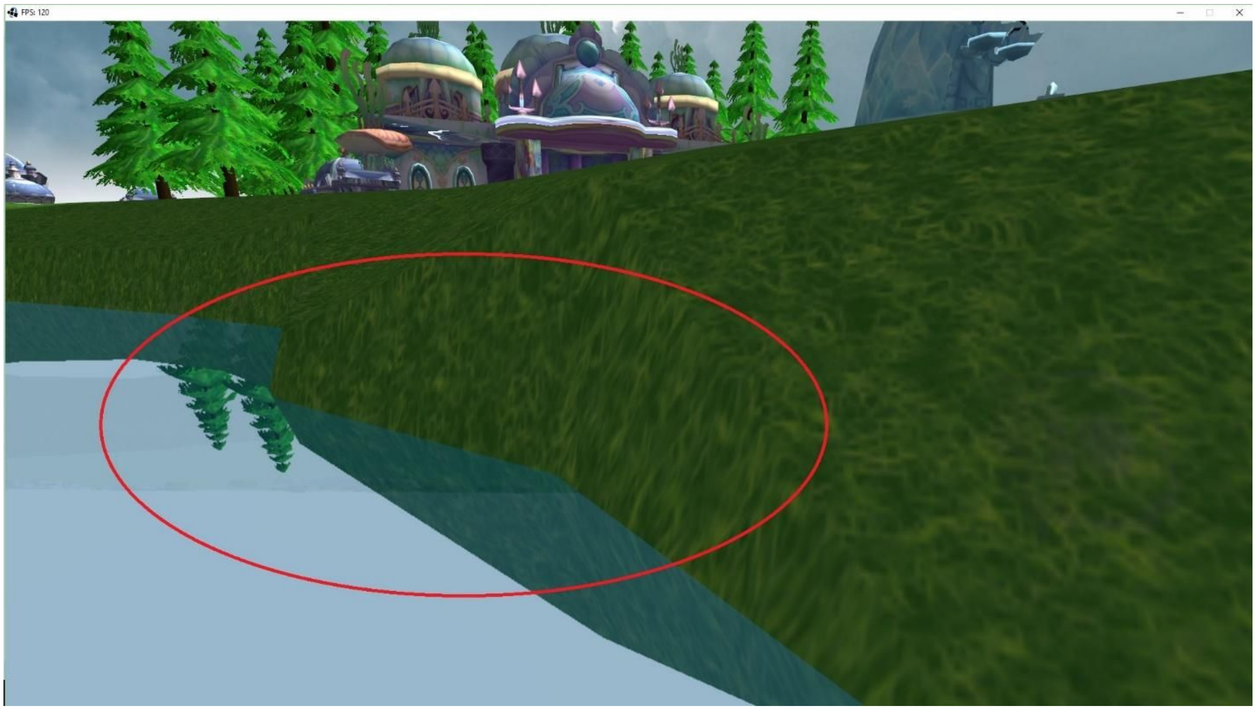
Softening the edges of the water

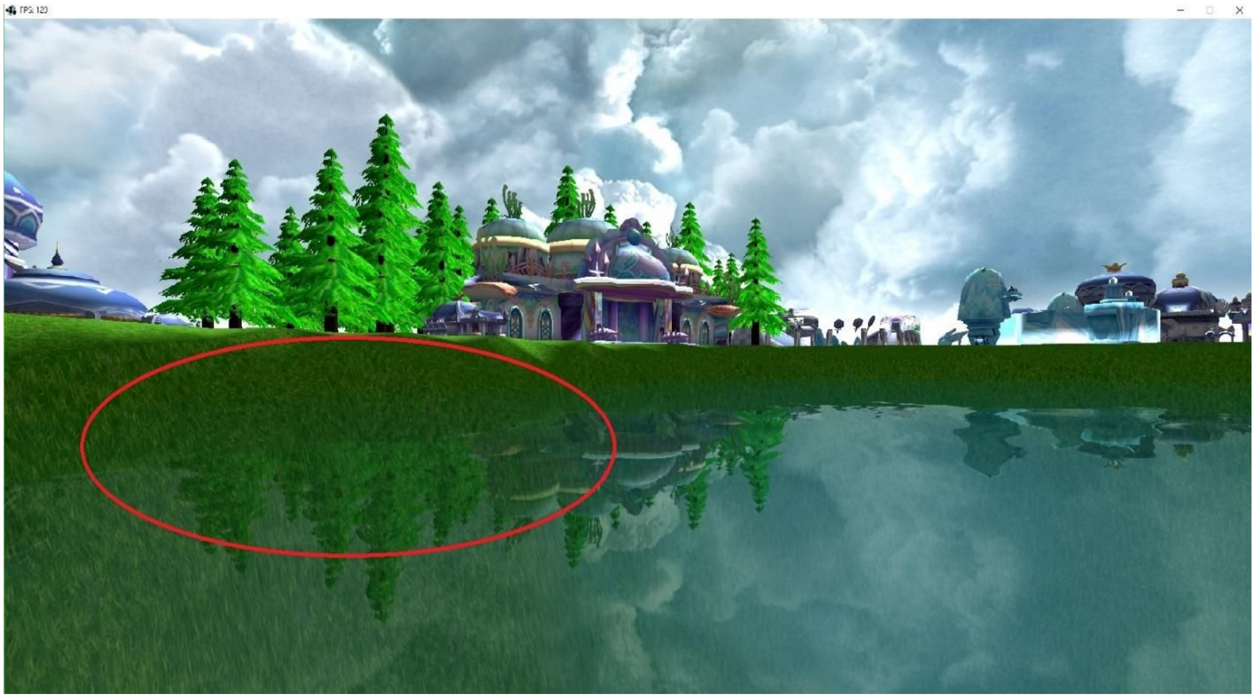
Figure 13

Finishing off the water by adding some soft edges before to start though that is using this new DUDV map and matching normal map so the specular highlights on the water match the distortion on the water as they should and it just looks a lot better and a lot more realistic than the setup that is shown in last part and these constants can be used in the fragment shader and this tiling value in the vertex shader if doing that then the water should now look something like this which may look a little bit too shiny right now but by the end of this part it will be looking pretty nice with those settings so the main thing that is finding the depth of the water and then using that to add some nice effects such as soft edges and we should also be able to fix that annoying glitchy edge to the water. To render the depth buffer of the refraction render pass to a texture using FBOS meaning that there are some information about the distance of the terrain and any other object under the water from the camera and having a visual representation of that depth texture here when applying to the water quad with projected texturing the whiter a pixel is the further away that point on the terrain is from the camera so for any given fragment on the water quad the refraction depth texture can show this distance the distance from the camera to the nearest object under the water there's also an inbuilt OpenGL variable which we can access in the fragment shader called GL_Frag_Cord this contains the window relative coordinate the current fragment and the Z component of this variable to give the depth of that fragment so again for any given pixel on the water quad the distance from the refraction depth texture can be got but this distance can be got from the GL_Frag_Cord variable if then to subtract one distance from the other this distance is left here which is the depth of the water from the camera perspective or more specifically the amount of water that's between the camera and the terrain or the object under water once this distance is got then to use it to add all sorts of depth effects to the water so in the water fragment shader the first thing that is to add a sampler for that refraction depth texture then in the water shader the usual stuff has to be done and get the location of the uniform so a new integer called location depth map will be created and then to just get the uniform location and spell depth map correctly and then to connect up the texture unit so it will indicate that sampling the depth map and sampling it from texture unit for then in the water renderer class to bind the depth map to texture unit four so binding to texture unit four and the depth map which can get from doing FBOS dots get refraction depth texture back in the water fragment shader that depth map is going to be sampled and we have to sample it up here before the refraction texture coordinates get distorted because we don't want the depth map to be distorted so to sample the depth map and sample it using the refraction texture coordinates because of course it was rendered during the refraction texture part and just take the r component the red component of that texture because that's where the depth information is stored unfortunately this doesn't get a distance this gets a number between zero and one which represents the distance that as mentioned earlier but it isn't even represented linearly either to expected the depth buffer value to increase linearly with distance like this but in fact it's actually something more like this giving a lot more depth precision to the nearer objects in the scene so it needs to convert this depth value into a distance so this conversion needs to know the near and far planes that is in the projection matrix.

Now to be loading these up as uniform variables. Now conversion and to calculate the floor distance. The distance from the camera to the terrain under the water and the distance needs to be calculated from the camera to the current fragment on the water's surface and that can be done as mentioned earlier using the GL Frank chord variable that OpenGL provides and we can use the Z component of that to get the depth and then again do the conversion to convert it to an actual distance so now having the floor distance here and the water distance here and as mentioned earlier the depth can be calculated of the water now by subtracting the water distance from the floor distance so let's now do that calculation so to calculate the water depth and do that by taking the floor distance and in the water distance so now getting the water depth and that can be used to do all sorts of cool effects but first that is going to be tested. it’s all worked by setting the output color off the fragment shader to that water depth and that need to make it into a vector four and also to need to make it a lot smaller because that’s a distance and a color is needed between 0 and 1 so to divide that by 50 just to make it a bit more visible and if you run that you should get something like this where the deeper areas of the water are white and the shallow areas especially the edges are shown in black so now that there is the information about the depth of the water we can use this to make the edges of the water much softer the easiest way is to just use alpha blending and to make the edges of the water quad gradually blend into transparency so back into the water fragment shader it's unnecessary to want a water to look like that and then in the water renderer, alpha blending needs to be enabled if we went to use it so in the prepare render method it needs to enable alpha blending and then after finishing rendering in the unbind method it will need to disable the alpha blending again and we can do that by calling GL disable and then GL lend so now go back into the water fragment shader and the alpha value of the output color needs to be set to determine how transparent the water should be so to set the out color dot a which is the alpha component. and for now just to set it to the actual water depth the way that the Alpha value is supported to vary with water depth should look something like this at the very edge of the water where the depth is zero. The water is supported to be completely transparent and so have an alpha value of zero as the water depth increases. The water is supported to become less transparent and then above a certain depth the water isn’t transparent at all and should render just like before at the moment. The alpha value just has been to be set to the water depth so there is something like this if dividing the water depth by a certain number assuming that is five then at that water depth so add a water depth of five the water has an alpha value of one and isn’t transparent at all however the alpha value should then stay at one for greater depths instead of increasing more so to clamp this value between zero and one so that anything above one gets clamped back down to one which is actually exactly what is supported if to change the depth at which the water has an alpha value of one then that five can be changed to whatever depth so now running that and the edges of the water are now nice and soft also the glitchy edges are now much improved but they are still importantly visible the reason for these glitches is the distortion that to add to the refraction and reflection textures before adding the distortion to the text is the reflection and refraction textures would stop exactly at the edges of the water but the distortion to these textures means that the cutoff point is no longer exactly in the correct place and it becomes visible on the water’s surface so if to lessen the distortion around the edges of the water, the effect of these qlitchy edges can be lessen so to reduce the distortion around the edges of the water to use the exact same code that is used for the transparency but to multiply it into the total distortion calculation here and this five is going to be change to a water depth of ten so that the transition period is a bit larger and a bit more gradual so if running that horrible edge glitch is now pretty much completely gone and that is going to be used for one more thing now and that is to dampen the specular highlights around the edge of the water because if these specular highlights are too strong then they ruin the soft edge effect that we have got so if running that it should be able to see that around the edge of the water the specular highlights get a lot dimmer and so they don’t ruin that soft edge effect. Another thing to improve the water is to use the normal vector that is calculated in the last part to assume that the water surface is completely flat for the calculation but it's not supported to see that the water to look completely flat last part so to generate customer normals from a normal map so those normals can be used in the frenetic calculation so to do this it first need to move the normal calculation up a bit so that it’s above the Finnell calculation and once that has been done, this normal can be used here instead of the vector 0 1 0 in the finale calculation if running that it has worked but it looks a little bit crazy a little bit too extreme and this is because the normals are all over the place at the moment making the water seemed a lot more bumpy than it’s perhaps realistic so to make the water up here a little bit by making the normals point more upwards as if the water were flatter and to do that to simply increase the Y component of the normals a bit if now running that the effect is much more subtle and actually looks pretty nice.

It is almost finished now but that nasty edge glitch is still occasionally visible so to try and get rid of it completely now in the main loop. there is something that should be remembered that to set up some clipping planes which make sure that the reflection texture only renders parts of the scene which are above the water and it cuts off exactly at the water’s surface this doesn’t leave much room for error and it’s what causes the glitches especially when the water was distorted now that the distortion has been dampened and the glitch is now almost gone but still the occasional pixel pokes through so instead of making the clipping plane cut off exactly at the water’s surface a little offset can be added to create a tiny overlap and to do this for the refraction texture as well if it needs to but it usually isn’t a problem if running that the glitch is now entirely gone however because an overlap has been added there will occasionally be things reflected in the water that shouldn’t be reflected the bigger offset but here there’s already a tiny little reflection here that shouldn’t be there so make sure that to keep that offset as small as possible so that is it for this.


FPS: 120


FPS: 120

Java Interfaces

A. Post-rendering Interface:

```
public class PostProcessing {

    private static final int size = 1;
    private static final float[] POS = { -size, size, -size, -size, size, size,
size, -size };
    private static VAO_Model quad;
    private static InverseChanger inverseChanger;


    public static void init(Parse parse){
        quad = parse.FillVAO(POS, 2);
        inverseChanger = new InverseChanger();

    }

    public static void ExePostProcessing(int tex) {
        start();
        inverseChanger.draw(tex);
        end();
    }


    public static void cleanUp(){
        inverseChanger.CleanUp();

    }

    private static void start(){
        GL30.glBindVertexArray(quad.getVaoID());
        GL20.glEnableVertexAttribArray(0);
        GL11.glDisable(GL11.GL_DEPTH_TEST);
    }

    private static void end(){
        GL11.glEnable(GL11.GL_DEPTH_TEST);
        GL20.glDisableVertexAttribArray(0);
        GL30.glBindVertexArray(0);
    }

}
```

B. The quad of Post-rendering:

```
public class InverseChanger {

    private TexDrawing renderer;
    private InverseSD shader;
    public InverseChanger() {
        shader = new InverseSD();
        renderer = new TexDrawing();
    }

    public void draw(int texture) {
        shader.begin();
        GL13.glActiveTexture(GL13.GL_TEXTURE0);
        GL11.glBindTexture(GL11.GL_TEXTURE_2D, texture);
        renderer.DrawShape();
        shader.end();
    }

    public void CleanUp() {
        renderer.cleanUp();
        shader.cleanUp();
    }

}
```

C. The shader of quad

```
import Shader.BaseShader;

public class InverseSD extends BaseShader {

    private static final String V_FILE = "/postProcessing/inverseVS.txt";
    private static final String F_FILE = "/postProcessing/inverseFS.txt";

    public InverseSD() {
        super(V_FILE, F_FILE);
    }

    @Override
    protected void getAllUniformIndexs() {
    }

    @Override
    protected void bindProperty() {
        super.bindProperty(0, "c_position");
    }

}
```

D. FBO of Post-Processing Effects

```
package postProcessing;

import java.nio.ByteBuffer;
```

```java
import org.lwjgl.opengl.Display;
import org.lwjgl.opengl.GL11;
import org.lwjgl.opengl.GL12;
import org.lwjgl.opengl.GL14;
import org.lwjgl.opengl.GL30;

import Draw.DrawSetting;

public class Fbo {

    public static final int NONE = 0;
    public static final int DEPTH_TEXTURE = 1;
    public static final int DEPTH_RENDER_BUFFER = 2;

    private final int width;
    private final int height;

    private int frameBuffer;

    private boolean multisample = false;

    private int colourTexture;
    private int depthTexture;

    private int depthBuffer;
    private int colourBuffer;

    /**
     * Creates an FBO of a specified width and height, with the desired type of
     * depth buffer attachment.
     *
     * @param width
     *            - the width of the FBO.
     * @param height
     *            - the height of the FBO.
     * @param depthBufferType
     *            - an int indicating the type of depth buffer attachment that
     *            this FBO should use.
     */
    public Fbo(int width, int height, int depthBufferType) {
        this.width = width;
        this.height = height;
        InitFBuffer(depthBufferType);
    }

    public Fbo(int width, int height) {
        this.width = width;
        this.height = height;
        this.multisample = true;
        InitFBuffer(DEPTH_RENDER_BUFFER);
    }
```

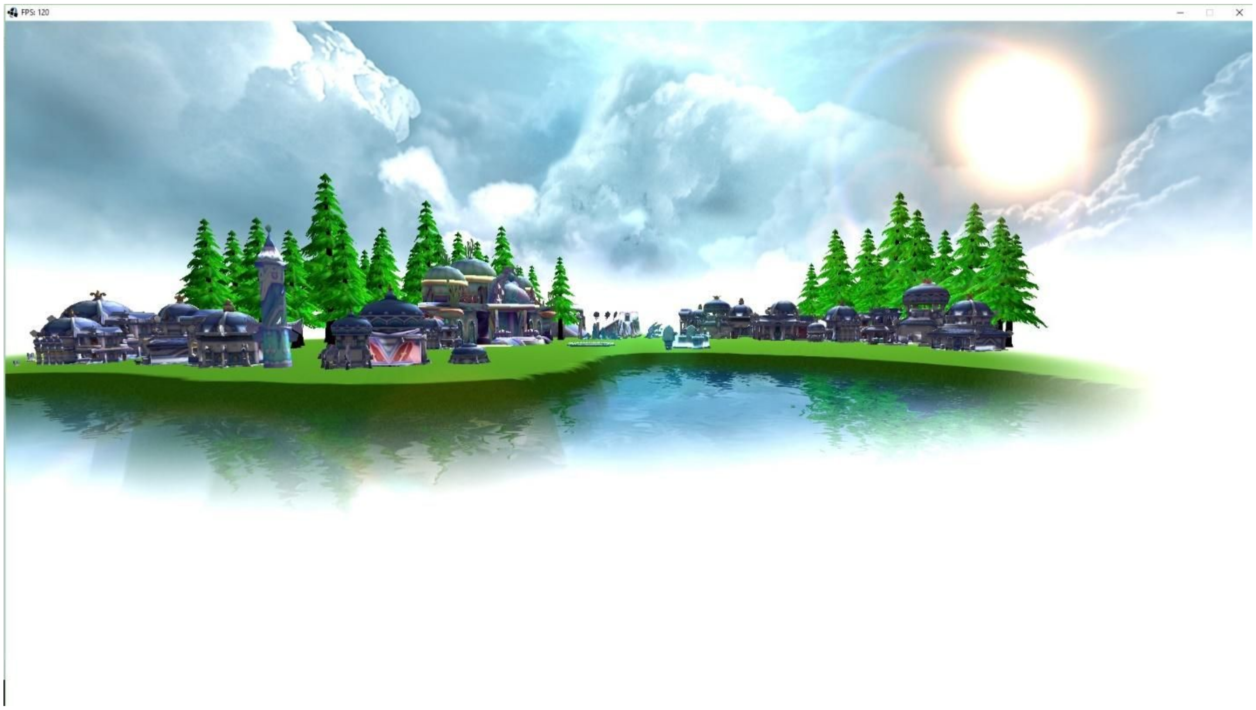
```
/**
 * Deletes the frame buffer and its attachments when the game closes.
 */
public void cleanUp() {
	GL30.glDeleteFramebuffers(frameBuffer);
	GL11.glDeleteTextures(colourTexture);
	GL11.glDeleteTextures(depthTexture);
	GL30.glDeleteRenderbuffers(depthBuffer);
	GL30.glDeleteRenderbuffers(colourBuffer);
}

/**
 * Binds the frame buffer, setting it as the current draw target. Anything
 * rendered after this will be rendered to this FBO, and not to the screen.
 */
public void bindFBuffer() {
	GL30.glBindFramebuffer(GL30.GL_DRAW_FRAMEBUFFER, frameBuffer);
	GL11.glViewport(0, 0, width, height);
}

/**
 * Unbinds the frame buffer, setting the default frame buffer as the current
 * draw target. Anything rendered after this will be rendered to the
 * screen, and not this FBO.
 */
public void unbindFBuffer() {
	GL30.glBindFramebuffer(GL30.GL_FRAMEBUFFER, 0);
	GL11.glViewport(0, 0, Display.getWidth(), Display.getHeight());
}

/**
 * Binds the current FBO to be read from (not used in tutorial 43).
 */
public void bindToRead() {
	GL11.glBindTexture(GL11.GL_TEXTURE_2D, 0);
	GL30.glBindFramebuffer(GL30.GL_READ_FRAMEBUFFER, frameBuffer);
	GL11.glReadBuffer(GL30.GL_COLOR_ATTACHMENT0);
}

/**
 * @return The ID of the texture containing the colour buffer of the FBO.
 */
public int getColourTexture() {
	return colourTexture;
}

/**
 * @return The texture containing the FBOs depth buffer.
 */
public int getDepthTexture() {
	return depthTexture;
```

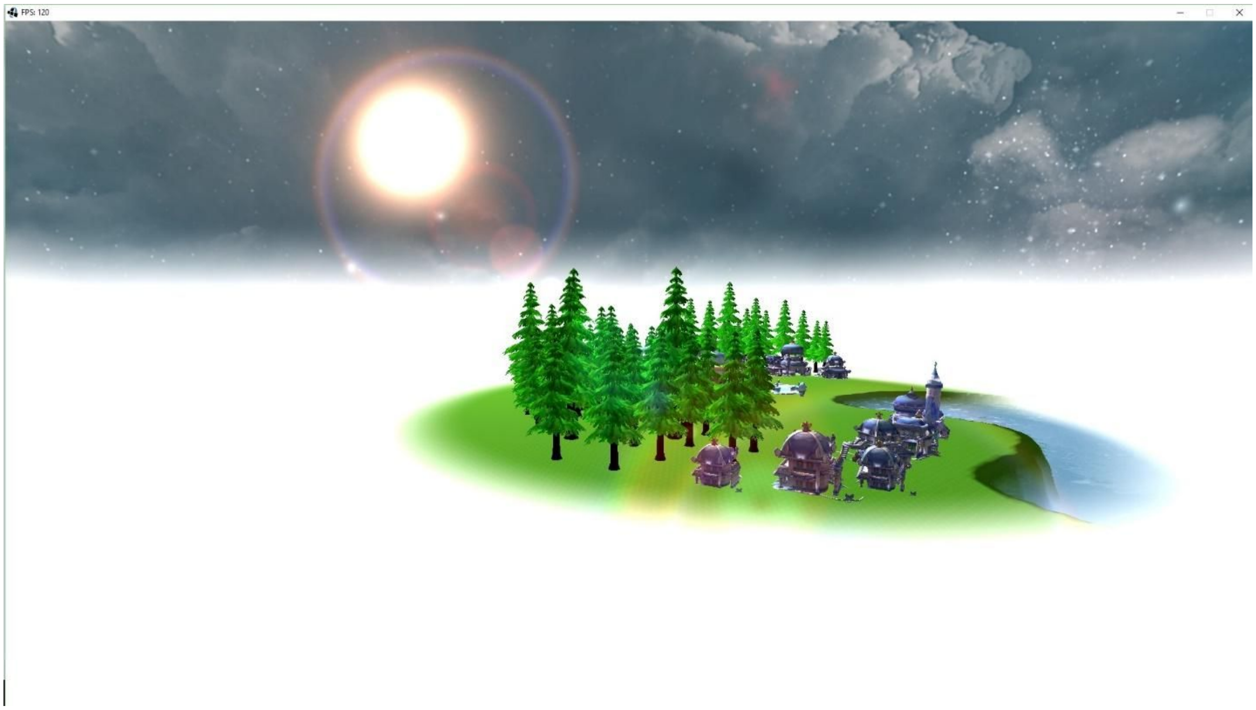
```
    }

    public void resolveToFbol(Fbo outputFbo) {
        GL30.glBindFramebuffer(GL30.GL_DRAW_FRAMEBUFFER, outputFbo.frameBuffer);
        GL30.glBindFramebuffer(GL30.GL_READ_FRAMEBUFFER, this.frameBuffer);
        GL30.glBlitFramebuffer(0, 0, width, height, 0, 0, outputFbo.width,
outputFbo.height,
                GL11.GL_COLOR_BUFFER_BIT | GL11.GL_DEPTH_BUFFER_BIT,
GL11.GL_NEAREST);
        this.unbindFBuffer();
    }

    public void resolveToScrren() {
        GL30.glBindFramebuffer(GL30.GL_DRAW_FRAMEBUFFER, 0);
        GL30.glBindFramebuffer(GL30.GL_READ_FRAMEBUFFER, this.frameBuffer);
        GL11.glDrawBuffer(GL11.GL_BACK);
        GL30.glBlitFramebuffer(0, 0, width, height, 0, 0, Display.getWidth(),
Display.getHeight(),
                GL11.GL_COLOR_BUFFER_BIT, GL11.GL_NEAREST);
        this.unbindFBuffer();
    }

    /**
     * Creates the FBO along with a colour buffer texture attachment, and
     * possibly a depth buffer.
     *
     * @param type
     *            - the type of depth buffer attachment to be attached to the
     *            FBO.
     */
    private void InitFBuffer(int type) {
        createFBuffer();

        if(multisample) {
            createMultisampleColourAttachment();
        }else {
            createTexAttach();
        }

        if (type == DEPTH_RENDER_BUFFER) {
            createDBufferAttach();
        } else if (type == DEPTH_TEXTURE) {
            createDTexAttach();
        }
        unbindFBuffer();
    }

    /**
     * Creates a new frame buffer object and sets the buffer to which drawing
     * will occur - colour attachment 0. This is the attachment where the colour
     * buffer texture is.
     *
```

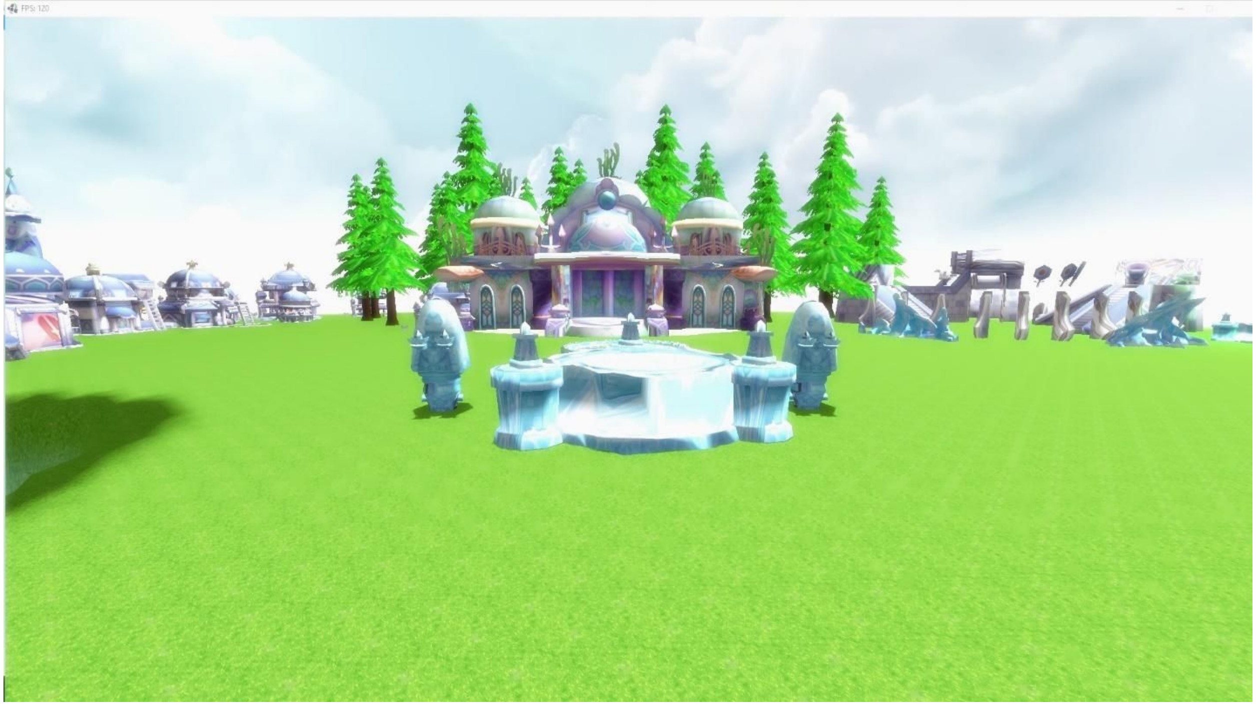
```
        */
       private void createFBuffer() {
             frameBuffer = GL30.glGenFramebuffers();
             GL30.glBindFramebuffer(GL30.GL_FRAMEBUFFER, frameBuffer);
             GL11.glDrawBuffer(GL30.GL_COLOR_ATTACHMENT0);
       }

       /**
        * Creates a texture and sets it as the colour buffer attachment for this
        * FBO.
        */
       private void createTexAttach() {
             colourTexture = GL11.glGenTextures();
             GL11.glBindTexture(GL11.GL_TEXTURE_2D, colourTexture);
             GL11.glTexImage2D(GL11.GL_TEXTURE_2D, 0, GL11.GL_RGBA8, width, height,
0, GL11.GL_RGBA, GL11.GL_UNSIGNED_BYTE,
                          (ByteBuffer) null);
             GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_MAG_FILTER,
GL11.GL_LINEAR);
             GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_MIN_FILTER,
GL11.GL_LINEAR);
             GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_WRAP_S,
GL12.GL_CLAMP_TO_EDGE);
             GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_WRAP_T,
GL12.GL_CLAMP_TO_EDGE);
             GL30.glFramebufferTexture2D(GL30.GL_FRAMEBUFFER,
GL30.GL_COLOR_ATTACHMENT0, GL11.GL_TEXTURE_2D, colourTexture,
                          0);
       }

       /**
        * Adds a depth buffer to the FBO in the form of a texture, which can later
        * be sampled.
        */
       private void createDTexAttach() {
             depthTexture = GL11.glGenTextures();
             GL11.glBindTexture(GL11.GL_TEXTURE_2D, depthTexture);
             GL11.glTexImage2D(GL11.GL_TEXTURE_2D, 0, GL14.GL_DEPTH_COMPONENT24,
width, height, 0, GL11.GL_DEPTH_COMPONENT,
                          GL11.GL_FLOAT, (ByteBuffer) null);
             GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_MAG_FILTER,
GL11.GL_LINEAR);
             GL11.glTexParameteri(GL11.GL_TEXTURE_2D, GL11.GL_TEXTURE_MIN_FILTER,
GL11.GL_LINEAR);
             GL30.glFramebufferTexture2D(GL30.GL_FRAMEBUFFER,
GL30.GL_DEPTH_ATTACHMENT, GL11.GL_TEXTURE_2D, depthTexture, 0);
       }

       /**
        * Adds a depth buffer to the FBO in the form of a draw buffer. This can't
        * be used for sampling in the shaders.
        */
```

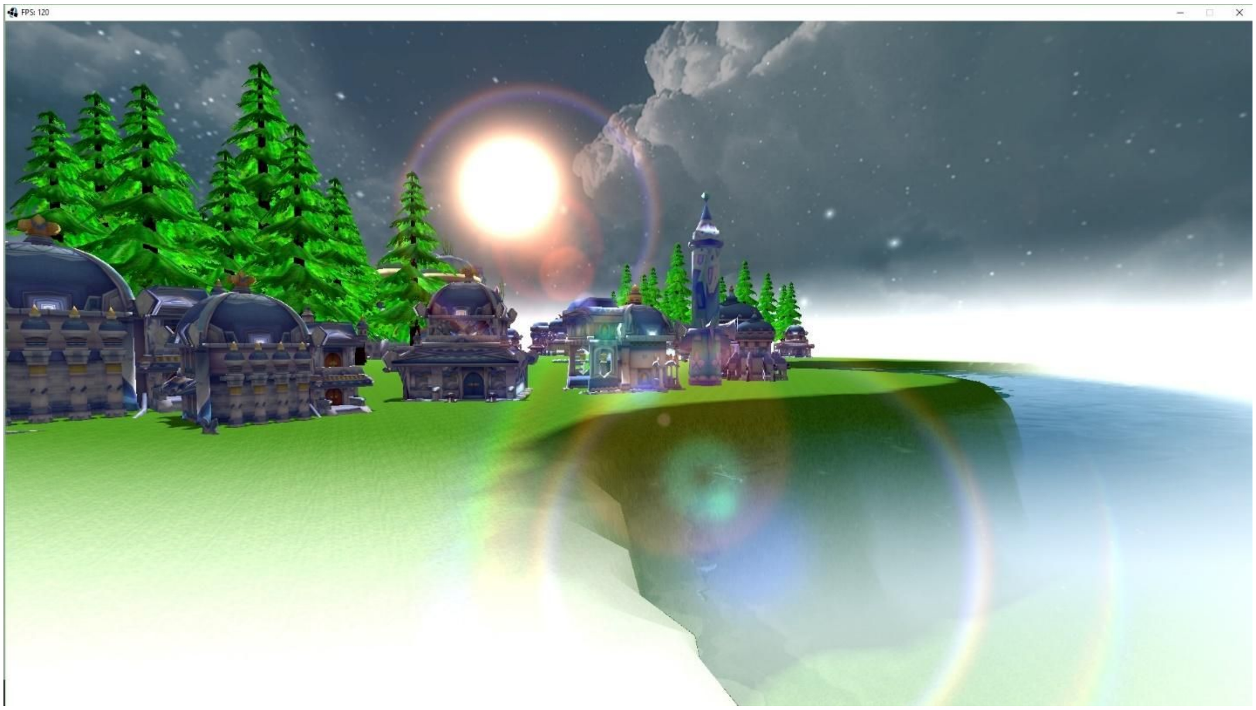
```
    private void createDBufferAttach() {
        depthBuffer = GL30.glGenRenderbuffers();
        GL30.glBindRenderbuffer(GL30.GL_RENDERBUFFER, depthBuffer);
        if(!multisample) {
            GL30.glRenderbufferStorage(GL30.GL_RENDERBUFFER,
GL14.GL_DEPTH_COMPONENT24, width, height);
        }else {
            GL30.glRenderbufferStorageMultisample(GL30.GL_RENDERBUFFER, 4,
GL14.GL_DEPTH_COMPONENT24, width, height);
        }
        GL30.glFramebufferRenderbuffer(GL30.GL_FRAMEBUFFER,
GL30.GL_DEPTH_ATTACHMENT, GL30.GL_RENDERBUFFER,
                depthBuffer);
    }

    private void createMultisampleColourAttachment() {
        colourBuffer = GL30.glGenRenderbuffers();
        GL30.glBindRenderbuffer(GL30.GL_RENDERBUFFER, colourBuffer);
        GL30.glRenderbufferStorageMultisample(GL30.GL_RENDERBUFFER, 4,
GL11.GL_RGBA8, width, height);
        GL30.glFramebufferRenderbuffer(GL30.GL_FRAMEBUFFER,
GL30.GL_COLOR_ATTACHMENT0, GL30.GL_RENDERBUFFER, colourBuffer);
    }

}
```

Demo






FPS: 120


FPS: 120

Conclusion

In this thesis, the rendering pipeline of the post-processing effects as a global overview concept of the graphics applications is presented. For this purpose of VAO and VBO are used, which are an important topic throughout the whole thesis.

The main part of this paper was to describe the concepts and design solution behind the implemented post-processing system. A user can then take such simple parts and combine them a new rendering effect. The purpose of the post-processing system is to improve the quality of the graphic engine.

Knowledgment

Many thanks to Dr.Yanjun Zhao. The subject of this thesis was challenging to me. She made this experience really valuable for me. Thank you so much!

References

[1] AKENINE-M¨OLLER, T., MUNKBERG, J., AND HASSELGREN, J.2007. Stochastic rasterization using time-continuous triangles.In Graphics Hardware, Eurographics Association, Aire-la-Ville,Switzerland, 7–16.

[2] BOULOS, S., LUONG, E., FATAHALIAN, K., MORETON, H.,AND HANRAHAN, P. 2010. Space-time hierarchical occlusionculling for micropolygon rendering with motion blur. In High Performance Graphics, Eurographics Association, Aire-la-Ville,Switzerland, 11–18.

[3] BOUVIER-ZAPPA, S., OSTROMOUKHOV, V., AND POULIN, P.2007. Motion cues for illustration of skeletal motion capture data. In NPAR, ACM, New York, NY, USA, 133–140.

[4] BRUNHAVER, J. S., FATAHALIAN, K., AND HANRAHAN, P.2010. Hardware implementation of micropolygon rasterization with motion and defocus blur. In High Performance Graphics,Eurographics Association, Aire-la-Ville, Switzerland, 1–9.

[5] COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributedray tracing. In SIGGRAPH, ACM, New York, NY, USA,vol. 18, 137–145.

[6] EGAN, K., TSENG, Y.-T., HOLZSCHUCH, N., DURAND, F., AND RAMAMOORTHI, R. 2009. Frequency analysis and sheared reconstruction for rendering motionblur. In SIGGRAPH, 93:1–:13.

[7] EPIC GAMES, 2010. MotionBlur post process effect. Unreal Developer Network, Unreal Engine 3 Documentation,http://udn.epicgames.com/Three/MotionBlur.html.

[8] FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K.,MARK, W. R., AND HANRAHAN, P. 2009. Data-parallel rasterization of micropolygons with defocus and motion blur. In High Performance Graphics, ACM, New York, NY, USA, 59–68.

[9] GRANT, C. W. 1985. Integrated analytic spatial and temporal antialiasing for polyhedra in 4-space. SIGGRAPH (July), 79–84.

[10] GREEN, S., 2003. Stupid OpenGL shader tricks. Talk at Game Developers Conference.

[11] GRIBEL, C. J., DOGGETT, M., AND AKENINE-M¨O LLER, T.2010. Analytical motion blur rasterization with compression. In High Performance Graphics, Eurographics Association, Aire-la- Ville, Switzerland, 163–172.

[12] GRIBEL, C. J., BARRINGER, R., AND AKENINE-M¨O LLER, T. 2011. High-quality spatio-temporal rendering using semianalytical visibility. In SIGGRAPH, ACM, New York, NY, USA, 54:1–54:12.

[13] HARGREAVES, S. 2004. In Detail Texture Motion Blur, W. Engel,Ed. Charles River Media, ch. 2.11, 205–214.

[14] HERZOG, R., EISEMANN, E., MYSZKOWSKI, K., AND SEIDEL,H.-P. 2010. Spatio-temporal upsampling on the GPU. In I3D2010, ACM.

[15] JONES, N., AND KEYSER, J. 2005. Real-time geometric motion blur for a deforming polygonal mesh. In Computer Graphics International 2005, IEEE Computer Society, Washington, DC, USA, 26–31.

[16] KASYAN, N., SCHULZ, N., AND SOUSA, T., 2011. Secrets of CryENGINE 3 graphics technology. SIGGRAPH 2011 Talks.

[17] LEHTINEN, J., AILA, T., CHEN, J., LAINE, S., AND DURAND,F. 2011. Temporal light field reconstruction for rendering distribution effects. ACM Trans. Graph. 30, 4.

[18] LOTTES, T., 2009. FXAA, February.

[19] FXAA WhitePaper.LOVISCACH, J. 2005. Motion blur for textures by means of anisotropic filtering. In EGSR.

[20] MAX, N. L., AND LERNER, D. M. 1985. A two-and-a-half-D motion-blur algorithm. In SIGGRAPH, ACM, New York, NY, USA, 85–93.

[21] MAX, N. 1990. Polygon-based post-process motion blur. Visual Computer 6 (November), 308–314.

[22] MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D.2010. Hardware-accelerated stochastic rasterization on conventional GPU architectures. In High Performance Graphics.

[23] MUNKBERG, J., CLARBERG, P., HASSELGREN, J., TOTH, R.,SUGIHARA, M., AND AKENINE-M¨O LLER, T. 2011. Hierarchical stochastic motion blur rasterization. In High Performance Graphics, ACM, New York, NY, USA, 107–118.

[24] OBAYASHI, S., KONDO, K., KONMA, T., AND IWAMOTO, K.-I. 2005. Non-photorealistic motion blur for 3d animation. InSIGGRAPH Sketches, ACM, New York, NY, USA.

[25] OVERBECK, R. S., DONNER, C., AND RAMAMOORTHI, R. 2009.Adaptive Wavelet Rendering. SIGGRAPH Asia 28, 5, 1–12.

[26] PEPPER, D. 2008. Per-pixel motion blur for wheels. In ShaderX6,W. Engel, Ed. Charles River Media, ch. 3.4, 175–188.

[27] POTMESIL, M., AND CHAKRAVARTY, I. 1983. Modeling motion blur in computer-generated images. In SIGGRAPH, ACM, New York, NY, USA, 389–399.

[28] RITCHIE, M., MODERN, G., AND MITCHELL, K. 2010. Split second motion blur. In SIGGRAPH Talks, ACM, New York, NY, USA, 17:1–17:1.

[29] ROSADO, G. 2007. Motion blur as a post-processing effect. In GPU Gems 3. Addison Wesley, ch. 27, 575–581.

[30] SAWADA, Y., 2007. Talk at Game Developers Conference CEDEC.SEN, P., AND DARABI, S. 2011. On Filtering the Noise from the Random Parameters in Monte Carlo Rendering. ACM Transactions on Graphics (TOG) to appear.

[31] SHIMIZU, C., SHESH, A., AND CHEN, B. 2003. Hardware accelerated motion blur generation. Eurographics.

[32] SHIRLEY, P., AILA, T., COHEN, J., ENDERTON, E., LAINE, S. LUEBKE, D., AND MCGUIRE, M. 2011. A local image reconstruction algorithm for stochastic rendering. In I3D, ACM, New York, NY, USA, 9–14.

[33] SOUSA, T., 2008. Crysis next gen effects. Talk at Game Developers Conference.

[34] SOUSA, T., 2011. CryENGINE 3 rendering techniques, August.Talk at Microsoft Game Technology conference Gamefest 2011.

[35] SUNG, K., PEARCE, A., AND WANG, C. 2002. Spatial-temporal antialiasing. IEEE TVCG 8 (April), 144–153.

[36] TATARCHUK, N., BRENNAN, C., ISIDORO, J., AND VLACHOS, A. 2003. Motion blur using geometry and shading distortion. In ShaderX2, W. Engel, Ed. Charles River Media.

[37] VLACHOS, A., 2008. Post processing in the Orange Box, February. Talk at Game Developers Conference 2008.

[38] ATTENE, M., KATZ, S., MORTARA, M., PATANE, G., SPAGNUOLO, M., AND TAL, A. 2006. Mesh Segmentation - A Comparative Study. In Proc. SMI.

[39] 3D Graphics Files, URL: http://www.wotsit.org

[40] BUNKER, M., ECONOMY, R., AND HARVEY, J. 1984. Cell texture—Its impact on computer image generation. In Proceedings of the Sixth Interservice/Industry Training Equipment Conference (National Security Industrial Association, Washington, DC, Oct.), 149–155

[41] Paar, P. (2006). Landscape visualizations: Applications and requirements of 3D visualization software for environmental planning. Computer, Environment and Urban Systems, 30(6), 815–839.

[42] Steven Wijgerse. Generating realistic city boundaries using two-dimensional perlin noise. University of Twente, 2007.

[43] Thomas Heath. A History of Greek Mathematics. Clarendon Press, 1921. Dover Reprint, 1981. 1.2, 17, 9, 11

[44] J. Groff, "An intro to modern opengl. chapter 1: The graphics pipeline." http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-TheGraphics-Pipeline.html, 2016. [Online; Last Date Accessed: 25-March-2016].

[45] Kusuma Agusanto, Li Li, Zhu Chuangui, and Ng Wan Sing. Photorealistic rendering for augmented reality using environment illumination. In Proceedings of the IEEE International Symposium on Mixed and Augmented Reality, ISMAR '03, pages 208–218, Washington, DC, USA, 2003. IEEE Computer Society

[46] Okan Arikan, David A. Forsyth, and James F. O'Brien. Fast and detailed approximate global illumination by irradiance decomposition. In ACM SIGGRAPH 2005, pages 1108– 1114, New York, NY, USA, 2005. ACM

[47] Carsten Benthin, Ingo Wald, and Philipp Slusallek. A scalable approach to interactive global illumination. Computer Graphics Forum, 22(3):621–631, 2003.

[48] Robinson, D. et al., 2014. High-Resolution Imagery and Video from SkySat-1. Louisville, KY, Proc. Joint Agency Commercial Imagery Evaluation. Skybox Imaging, 2015. Skybox Imaging. [Online] Available at: http://www.skyboximaging.com/

[49] J. Kautz, P.-P. Vazquez, W. Heidrich, and H.-P. Seidel. A Unified Approach to Prefiltered Environment Maps. In 11th Eurographics Rendering Workshop 2000, pp. 185-196

[50] Australian Associated Press, "PNG leader apologises to Bougainville for bloody 1990s civil war", 29 January 2014 https://www.theguardian.com/world/2014/jan/29/papua-new-guinea-apologises-bougainville-civil-war

[51] 3Dlabs. “Permedia 3: Inside the Third Generation of Desktop 3D.” Proceedings of Hot Chips 10, 1998, 233-243.