Author: Callan Moore

Technique Report

TECHNIQUES IMPLEMENTED

- ➔ Animated water
- ➔ Reflections

The two techniques that I have implemented in my graphics summative are intrinsically linked together so I will explain the fundamental steps into created the combined effect.

SETUP

The First step I had to do was to setup some background variables, functions and states that could be called to implement the chosen techniques.

Creating the Texture Resource Class

I first created an entirely new class 'TextureResource' which contains a Texture2D graphics resource and two different views to it. The RenderTargetView for directly rendering to the texture via the graphics device. Also created a ShaderResourceView so that I can pass the texture into a shader file to be read.

The functionality of the class required it to be able to set itself as the current RenderTarget on the graphics device as well as clear the Texture to the Renderers clear color. The final functionality is to return the ShaderResourceView when asked by another object.

I gave The TextureResource class the graphics device as it related directly and only to graphics and requires some direct functionality of the device. This also allows the Renderer to create TextureResource without the class needed to know about the Renderer in return and avoids circular includes. I therefore use the Renderer to create the TextureResource class and initialise the Texture2D component binding it with both D3D10_BIND_RENDER_TARGET and D3D10_BIND_SHADER_RESOURCE so that both views can be created referencing the same texture. Otherwise the rest of the initialisation of TextureResource components is standard.

The Application class is my main game loop and holds all the objects for processing and renderering. I create two instances of the TextureResource class, one to hold a texture for refraction and the other to hold a texture for reflection.

Add an additional Rasterizer State to the Renderer

I created an additional rasterizer state to change the 'front counter clockwise' culling. Normally this value is set to false to create the correct culling for clockwise winding order polygons. I set this value to false because I will use this state for rendering a reflection that has been reflected across a particular plane. This inverts the polygons and to have the correct culling this value must also be inverted.

I created a simple function on the Renderer to set this state as the current Rasterizer state.

Extra Functionality for the Renderer

Two additional functions were created in the renderer in order for my techniques to be implemented. The first was a reset the RenderTarget of the device to the back buffer of the swap chain. During both the Refraction and Reflection Texture renderering the RenderTarget is changed so this needs to be able to be set back to default.

The second was to reflect all the currently active lights in the scene across the plane of reflection so that the reflections are rendered with the correct lighting. This is simple done by creating a reflection matrix (A function that takes in a plane and calculates the reflection matrix. In the DX10_Utilities file) and multiplying by each active light individually.

Creating the Water Object and Shader

To generate the animated water with refraction and reflection I created an entirely new Object and Shader called 'Water'. The Shader file itself will be explained in detail in the following sections. The object is derived from my generic object with only a little bit of extra functionality. The Object must have a Water Shader class and a rippleScale which is used to control how much the water distorts and simulates a rippling effect.

The Water Shader class is a container and interface to the water.fx file that takes in all inputs necessary from the Water Object and passes them all to the GPU and calls upon the Waters mesh to render using the standard technique.

Updating the current Object and Shader files to handle Refraction and Reflection

The way my objects and shaders work is each type of object has its own shader class and file. This created a little bit of difficulty for these techniques as the refraction and reflection is created by rendering all other objects in the scene but the water in slightly different ways.

Firstly in my LitTex Shader class I had to create two new techniques pointers so that I could call upon the techniques that I would create in the shader file itself. I called the techniques 'RefractTech' and 'ReflectTech' and had to add an additional variable to pass though. The new variable was a plane that would be used by both techniques though with different uses which will be explained in the sections below.

The change was in the Object itself was the Render() call had to take in the plane so that it could be passed along. Also if the Reflection technique was the technique the object was going to be rendered with the object would calculate the reflection matrix using the plane passed in and reflect its own world matrix to give a reflection from the cameras point of view. Then the object would pass through the necessary variables including the new world matrix to the shader so that it could be rendered.

Creating the actual Instance of the Water

I create the instance of the water object in the Application and initialise it with a reference to the Renderer, a simple plane mesh, a reference to the Water Shader and a texture that represents a normal map rather than an actual texture. After the initialisation I set the object to be a scrolling object with a speed of 40 (takes 40 seconds to translate the texture cords back to their original position) and a direction which scrolls directly on the y axis only.

Then I set a rippleScale to control how much distortion the water creates when simulating ripples, the higher the factor the more distortion (simulating rougher water).

GENERATING THE WATER EFFECT

The process of creating the water requires 3 steps.

- ➔ Render a Refraction Texture
- ➔ Render a Reflection Texture
- ➔ Combine the Refraction and Reflection textures with the waters Normal map and Render

Render the Refraction Texture

Before the Refraction Shading

The first step to creating a Refraction texture is to create a clip plane that will clip all pixels drawn above the water. The clip plane the plane of the water but inversed so that the clipping occurs above the surface because refraction is only concerned with what is underneath (visible through the water). I add an addition value to the d component of the plane to raise the height of the water a little to stop the artifacting that occurs with translating the normal of the water's edge creating a horrible line of back buffer color.

The next step involves the first TextureResource I created. I tell the Refraction Texture Resource to set its Render Target View as the Render Target of the device and clear it so it is ready for input. This means that any renderering the device does is now to the Refraction Texture instead of the usual back buffer.

Now I render each object that has at least part of itself underneath the water using the Refraction Technique I set up in the shader earlier. This involves passing in an addition plane to be used for clipping pixels above the water line. The objects render call packages all variables needed to be sent to the shader and passes them through to the Shader class to be transferred onto the GPU and rendered using the refraction technique.

The technique uses the following Vertex and Pixel shader to create the refraction.

Refraction Vertex Shader

The refraction vertex shader is fairly generic. I take the vertex position and homogenise it and transform the position into world space by multiplying it by the world, view and projection matrices.

At this stage I don't manipulate the texture coordinate, I just pass it directly through to the Pixel shader.

I calculate the normal vector of the vertex by also multiplying the normal of the vertex by the world matrix and normalise it for use in the Pixel Shader.

All the above is standard Vertex Shader code and the real difference is I calculate a clip distance to check if the vertex position is above the water surface. The pixel shader will clip any pixel that is above the water surface as it is not necessary in the refraction.

Refraction Pixel Shader

The pixel shader has no difference from the usual pixel shader except that it must take in a input structure that also includes the clip distance.

After the Refraction Shading

Lastly to complete the refraction I reset the graphics device Render Target back to the default back buffer on the swap chain.

Render the Reflection Texture

Before the Reflection Shading

Just like the refraction texture I get the TextureResource specially created to render a reflection to. As before I get the Reflection TextureResource to set its RenderTargetView as the RenderTarget on the graphics device and clear it so it's ready for input.

Similar to the refraction again I create a plane. This plane will be used to create a reflection matrix from instead of clipping. The reflection plane is the exact plane equation of the water.

Using the reflection plane I reflect all the active lights around the plane using the Renderer function I created in the setup.

Now the device and objects are ready for renderering and I render all objects that can be reflected into the water setting the technique to 'Reflect' and pass in the reflection plane.

In the Objects render function I create a reflection matrix from the plane and reflect the objects world matrix to create a flipped view of the object that will act as the reflection in the water. At this stage I also apply the reflection rasterizer state that I created on the Renderer so that the reflection is rendered with the correct faces and culling. Then I bundle all the object variables as usual and pass them to the Shader class for transferring onto the GPU before running the shader effects file.

The technique uses the following Vertex and Pixel shader to create the reflection.

Reflection Vertex Shader

The reflection technique uses the standard vertex shader of Littex shader file which just transforms the vertex into world space and passes through the vertex information to the Pixel Shader.

Reflection Pixel Shader

The pixel shader first calculates the pixel position distance from the plane. Samples the texture map to return the diffuse color and does the same for the specular map.

Once that colors have been sampled the distance from the reflection plane is checked, if the result is greater than zero the pixel is above the water and therefore does not need ot be rendered. I force a clip here by clipping if the diffuse alpha is any number less than the entire range. If the clip happens it stops the pixel shader processing and can save valuable time.

If the pixel passes the distance test and is part of the reflection I calculate the specular power and light the pixel as usual returning the lit color of the pixel.

After the Reflection Shading

To complete the reflection I have to reflect the lights back across the same plane to put them back in their correct places for any other technique. I reset the graphics device Render Target back to the default back buffer on the swap chain so that I no longer render to the reflection texture.

Rendering the Water

Before the Rendering

The process of the water object (calls the DX10_Obj_Generic BaseProcess) calculates the new world matrix just like any other object. It also calculates the texture coordinate translation to create a scrolling texture effect. The calculation takes the time elapsed divided by the scrolling speed to map a 0-1 value. I then scale the scroll direction by the result and that gives the new translation of the texture coordinates.

The Render function of the water just takes in both the Refraction and Reflection Textures as ShaderResourceViews and then bundles the water variables including the rippleScale and texture coordinate translation. The package of variables is then sent to the Water Shader class to be placed in the GPU before executing the shader effects file.

Water Vertex Shader

The water vertex shader does the usual vertex position transformation to world space passes the texture coordinates on.

This is where the magic starts to happen. I need to calculate a refraction and reflection position. I calculate a matrix from the View, Projection and world matrices in that order and multiply the vertex position by this matrix to get the position of the refraction and reflection in world space for their own textures.

Then I output to the Pixel Shader

Water Pixel Shader

The pixel shader takes the vertex texture coordinates and translates them by the given texture translation vector to simulate moving (scrolling) water.

The texture coordinates for the reflection and refraction textures are both calculated using the same equation and stored in a ’rippleTexCoord’ variable. The calculation converts the positions into texture coordinates within a -1 to 1 range.

I sample the normal map of the Water to get a normal vector (not a color) and convert it into the range of -1 to 1 as well.

Then I re-position the rippleTexCoord by multiplying the normal vector by the rippleScale to control the size of the ripple and add it to the current texture coordinate. This creates the distortion effect that simulates rippling water by moving the position to a slightly off position.

Now I can use the rippleTexCoord to sample the reflection and refraction textures and get their color values. In order to give the water a slight bluish tinge so that is more clearly distinguishable as water I add a small value to the blue channel of the refraction color.

Lastly I linear interpolate between the two sampled colors to create a blend of the color that leans towards the refraction color and return the final result.

After the Rendering

The only clean up I need to do after the rendering is in the Water Shader I set apply the Refraction and Reflection ShaderResourceViews on the GPU to NULL so that there is no issues when I try to write to them the next frame.