**Tutorial 18: Large Terrain Rendering**

http://web.archive.org/web/20140722211759/http:/rastertek.com/pic3001.gif

In the last several years a number of graphics engines have been released that are rendering vast amounts of terrain. Some of the more recent ones are now rendering over 32km (20 miles) of viewable terrain. The ability to render this much viewable terrain has come mainly from the increased power in consumer level video cards. Otherwise (with the exception of deferred shading) the terrain rendering techniques have remained pretty much the same as they were in years previous.

Level of Detail

To render several kilometers of terrain you need to employ one of the various level of detail (LOD) techniques. LOD is a group of techniques that share the common goal of reducing the polygon count and complex shader use on distant terrain. If you attempt to render a large amount of terrain without reducing any of the detail you will quickly find out that the average video card will not be able to render it at a reasonable speed. However even a simple LOD implementation will allow you to draw huge amounts of terrain and at the same time maintain an impressive frame rate.

There are many different LOD algorithms that dynamically reduce distant terrain detail. However most of them cause terrible terrain "popping" effects that are very distracting to the eye as you approach distant terrain. As well these same dynamic algorithms generally have issues with terrain edges that don't align correctly causing noticeable cracks in the terrain. To avoid these two common problems with dynamic LOD algorithms we will instead cover a method that uses pre-built terrain which is free of these issues. The method we are going to use is called node based LOD.

Node Based LOD

Node based LOD is a method where the entire terrain is evenly broken up into multiple nodes. Each node is the same size and covers a specific distance such as 128 meters by 128 meters. Now what is most significant about this method is that each node has multiple quality levels. For example a node might have a low, medium, and high quality version which has been pre-built in a terrain editor. Then when we render that node we determine the distance it is from the camera and from that we pick which quality version to render. If it is close to the camera then we render the high quality version, if it is a bit further we render the medium quality, and if it is past a certain range but within the maximum render distance then we draw the low quality version. You can of course have as many quality levels as you like, but generally three or four qualities is sufficient.

The next part of node based LOD is that we use background threads to load and unload nodes as the camera moves around the terrain. Most video cards don't have enough memory to hold several kilometers of multiple quality terrain nodes so loading/unloading is necessary. And to keep our frame rate from being disrupted we need to use background threads to do the loading. For example as the camera moves closer to medium quality node it will need to load in the higher quality version on a thread. Once that is loaded it then makes the switch to rendering the high quality version. And finally it unloads the medium quality version it was using previously.

Note that in DirectX 11 you should not be creating and releasing vertex and index buffers to perform the loading and unloading of terrain nodes. Doing this will cause the video card to lock and will also cause memory thrashing. You should instead be using an array of pre-initialized dynamic vertex buffers for each quality level. These buffers are reused instead of being released and recreated. Since the buffers for each quality level are the same size you can just use the Map function to overwrite the old data. For example if you have nine high quality nodes then you should have an array of ten vertex and index buffers dedicated just to high quality nodes. As you need to load a new high quality node you use the tenth unused vertex/index buffer and Map the terrain node data into it and give the node a pointer to it. Then you mark the previous high quality vertex and index buffer that is no longer being rendered as unused with just a boolean flag.

For each quality level we also have a shader (or multiple shaders). The super complex shader you use for up close terrain should never be used for distant terrain. Also try to sort by shader when rendering and try to minimize constant buffer updates to get the best frame rate you can.

The final part of node based LOD is that we render the nodes according to the location of the camera. The node that the camera is currently in and the eight nodes around it must be rendered as high quality. This allows us to always keep an area that surrounds the camera in high quality and it also gives us enough time to load more high quality nodes as the camera moves. Note that high frequency effects such as normal mapped textures will not show up in the distance so there is no need to do more than nine nodes of high quality terrain.

The next 16 nodes that surround the 9 high quality nodes are rendered as medium quality. You could do even more than 16 (such as the next 24 nodes that surround those 16 nodes) in medium quality, but this is something that is completely flexible and is up to you to decide once you have tested things out.

Finally all the remainder of nodes that are rendered beyond that are done so in low quality. There can be exceptions of certain special nodes that we may want more distant detail on, but we'll discuss that later. Here is a diagram showing what I just explained:

Now even though the diagram only shows two levels of low quality nodes you should do many more. But first you need to determine a terrain budget to figure out exactly how many you should be drawing. We'll discuss the budget shortly but we need to first talk about node culling.

Node Culling

Using node based LOD allows us to very easily cull nodes that aren't viewable. Each node is a specific size at a specific location so we can do a quick frustum test again a cube shape to quickly cull nodes from being rendered. You can even create a quad tree to do the culling at a higher speed. On average at any given camera location you will never be viewing more than 30 percent of the terrain. For example the diagram below shows the nodes that are culled in grey if you are in the center looking upwards:

The reason I am discussing node culling now is because it relates heavily to the terrain budget. All of the culled surrounding nodes must still be loaded into memory, but our frame rate only reflects what we are drawing at that moment. So keep this in mind when creating your terrain budget.

Terrain Budget

Before embarking on rendering massive amounts of terrain we should probably do some planning in advance. The whole idea of measure twice and cut once can save us a lot of coding and help us go down the right direction from the start.

We want to determine as close as possible the amount of terrain we should be rendering based on our target system specifications. We also need to take into consideration what else is going to be rendered and what percentage our terrain is allowed to take up. Below is a quick example to help us make decisions about how much terrain to render.

 Rough Poly Budget = About 500,000 polygons.

 Memory Budget = Assuming target spec is 1GB video card, use about 500MB for terrain, remainder for trees, buildings, etc.

 Leave enough room in memory (about 10 percent) to provide a cushion in case other issues show up.

With this rough spec in mind we can now setup the node qualities:

High Quality Node

 Full Graphics: Normal Mapped with dynamic lighting and four layers of texture blending.

 128x128 \* 2 triangles = 32,768 polys

 Memory: 32768 polys \* 3 vertices \* 64 bytes (12 byte position \* 8 byte tex \* 12 byte normal \* 12 byte tangent \* 12 byte binormal \* 8 byte tex) = 6144 KB (6 MB)

 9 Nodes = 54 MB memory, 0 to 384 meter terrain coverage, 294,912 polys

 Textures: 4 terrain textures 512x512, 1-2 normal maps 512x512, 1 blending texture 512x512

Medium Quality Node

 Reduced Graphics: Color mapped with dynamic lighting only.

 64x64 \* 2 triangles = 8,192 polys

 Memory: 8192 polys \* 3 vertices \* 36 bytes (12 byte position \* 12 byte color \* 12 byte normal) = 864 KB (0.84 MB)

 16 nodes = 13.5 MB memory, 384 to 640 meter terrain coverage, 131,072 polys

 Textures: None

Low Quality Node

 Reduced Graphics: Color mapped with dynamic lighting only.

 16x16 \* 2 triangles = 512 polys

 Memory: 512 polys \* 3 vertices \* 36 bytes (12 byte position \* 12 byte color \* 12 byte normal) = 54 KB (0.05 MB)

 24 nodes = 1.25 MB memory, 640 to 896 meter terrain coverage, 12,228 polys

 Textures: None

Low Quality Node (second layer)

 Reduced Graphics: Color mapped with dynamic lighting only.

 16x16 \* 2 triangles = 512 polys

 Memory: 512 polys \* 3 vertices \* 36 bytes (12 byte position \* 12 byte color \* 12 byte normal) = 54 KB (0.05 MB)

 32 nodes = 1.69 MB memory - 896 to 1152 meter terrain coverage, 16,384 polys

 Textures: None

So this spec gives us the same terrain as we diagrammed earlier. But of course we want to add more low quality nodes to cover even more distance. To test an average polygon count I wrote a quick program to render just plain colored nodes but still use the vertex count for low, medium, and high quality nodes that we defined above.

As you can see it rendered the nodes the camera was over with higher polygon counts (blue), the more distant medium nodes with half the polygon count (green), and then the distant nodes (red) in low polygon mode. The results that when I rendered 16x16 node terrain (2km) I got on average about 250k polygons. When I increased the node count to 25x25 (3km) I got on average about 375k polygons. Also I was well over 1000fps on my good computer and just over 100fps on my weak laptop, so this gave me an indication of how far I could take things with the current spec and node quality setup I defined here.

Now also note that frame rates are just rough, but they give us a general idea. In fact most video card drivers don't push the card to maximum unless absolutely required so that they can conserve on energy. So you could in fact render even more nodes and notice your frame rate is even better. Though once again this is just a rough test to see what I should aim for to start with. And like always if we code things well enough it should be simple to change a couple variables and render more nodes or add new quality levels. All of these things will need to be tested before we decide on the end state of our quality levels and node counts.

Terrain Building

To build good terrain you generally need to write your own terrain editor. Especially since we are planning to export multiple quality levels for each terrain node it makes more sense to write a tool that will automatically do it for us.

For implementation the vast majority of terrain editors use Perlin noise to generate fractal terrain, and then they add numerous tools and other algorithms to the editor so that the terrain can be modified to achieve a specific look. Some terrain editors go even further and use biological algorithms to place plants, trees, river erosion, and so forth.

When building a terrain editor it is usually a good idea to use something quick like Visual C++ to quickly prototype a program. Remember that artists prefer things like sliders and dials instead of text and numeric input boxes, and Visual C++ has those built in already.

Terrain editors can be a huge project on their own, but for the time being it is sufficient to build something that can generate perlin noise and export it to your own model format. Build the export function to export three quality levels for each 128x128 meter section of the terrain. High quality would be a 129x129 portion of the height map. Medium quality would be the same 129x129 section shrunk down (preferably an edge preserving re-sample) to 65x65. And finally the low quality version would be a 17x17 height map that was sampled down from the 129x129 section.

One thing to keep in mind is that it is common to use just 8 bits to represent the height giving only a range of 0 to 255. This precision is extremely lacking isn't useful for creating large realistic terrain. It is generally better to go with a 16 bit height value instead even if you don't use the entire 16 bit range.

Implementation

Now that you have a simple terrain editor that can export to your specific terrain model format you can now start rendering the terrain nodes as I detailed above. There is no need for any new graphics code in this tutorial as the previous terrain tutorials have already shown you all the code you will need to complete this project by yourself. However there are some extra details that I would like to discuss.

The first thing is that some nodes always require at least a minimum medium quality even in far distance, for example mountains. So in the terrain rendering engine you should have a method of ensuring that some nodes can be flagged to never lower their quality below a certain limit.

The second thing is that maybe you found a limit of only being able to render 16km but have a 32km terrain. This is not an issue because it just involves loading more low quality nodes as they come into the 16km viewable distance and then unloading the low quality nodes that are outside of that distance. Remember once again to reuse vertex and index buffers even for low quality nodes.

In addition a trick that is used to bring distance nodes in smoothly (instead of popping into view as they get loaded at a distance) is to set the far view plane correctly. For example if our limitation is 16km viewable distance then set the far view plane at 15.8km. Then make sure that you do load nodes that come into the 16km range before they get to the 15.8km distance. Now when the node comes into range of being rendered they will be drawn smoothly polygon by polygon using the far plane culling. Once again we are always looking at ways to remove the popping that occured with traditional LOD methods.

Also since I used color maps as an important tool for medium and low quality nodes I would like to explain how I build them. I generally render a high quality node by itself from a top down perspective. I flatten the terrain and set the light to be straight down. At this point I take a screenshot and then shrink it down to the medium and low quality node sizes as a color map.

The final thing I want to add is don't be afraid to go really low detail with the shaders on nodes that are outside of the high quality distance. If you put a high quality node beside a medium quality node at a medium distance you will notice they barely look any different other than the lighting looks softer on the medium quality node. And considering you will be covering a lot of the terrain with buildings, trees, and so forth the lighting becomes even less noticeable.

Summary

So you now understand the basis behind node based LOD and how to implement it to draw several kilometers of terrain. The key to getting a massive range is to test and try pushing the limits with low quality nodes and optimizations around them. Also there wasn't any code for this tutorial since all the previous tutorials have already covered all the graphics techniques you need to render terrain using DirectX 11.

To Do Exercises

1. Create a terrain budget.

2. Create the terrain editor to build and automatically export Perlin noise generated terrain into the terrain model format with three quality levels.

3. Write the terrain rendering program as detailed above. Attempt first for about 4km of viewable terrain.

4. Update the program to render at least 8km of viewable terrain and another 8km that is not viewable so that 16km of terrain can be navigated.

**Tutorial 12: Perturbed Clouds**

http://web.archive.org/web/20140722210824/http:/rastertek.com/pic1001.gif

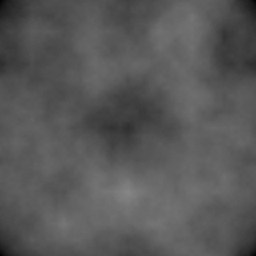
This DirectX 11 cloud tutorial is based on the previous bitmap clouds tutorial. We will be doing just some slight modifications to the previous tutorial code to implement perturbed clouds. Note that if you have already looked over and understand the fire tutorial in the DirectX 11 section then you already understand how to implement perturbed clouds. I recommend reading that tutorial if you want a more in-depth understanding of how the shader works.

In the previous tutorial we used two cloud textures and scrolled them at different speeds along the sky plane to simulate a basic cloud system. However as clouds move through the sky their formation modifies slightly. So for this tutorial we will just use a single cloud texture but then perturb the sampling of the texture by a secondary noise texture. By slightly offsetting where we sample the clouds from it gives the effect of the edges and insides of the clouds expanding and collapsing as they move over the sky plane.

The bitmap cloud texture that we will use is the following:



Then we will use a perlin style noise texture for the perturbed sampling:



By offsetting where we sample the cloud texture from using the noise texture as the perturbed coordinates we will get the desired effect of clouds that change their formation as they scroll over the sky plane.

Skyplane.vs

The vertex shader is the same as the previous tutorial.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplane.vs

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer MatrixBuffer

{

matrix worldMatrix;

matrix viewMatrix;

matrix projectionMatrix;

};

//////////////

// TYPEDEFS //

//////////////

struct VertexInputType

{

float4 position : POSITION;

float2 tex : TEXCOORD0;

};

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

};

////////////////////////////////////////////////////////////////////////////////

// Vertex Shader

////////////////////////////////////////////////////////////////////////////////

PixelInputType SkyPlaneVertexShader(VertexInputType input)

{

PixelInputType output;

// Change the position vector to be 4 units for proper matrix calculations.

input.position.w = 1.0f;

// Calculate the position of the vertex against the world, view, and projection matrices.

output.position = mul(input.position, worldMatrix);

output.position = mul(output.position, viewMatrix);

output.position = mul(output.position, projectionMatrix);

// Store the texture coordinates for the pixel shader.

output.tex = input.tex;

return output;

}

Skyplane.ps

The pixel shader is where we do all the work in this tutorial to create the perturbed cloud effect. The shader now takes a cloud and a noise texture as input textures. The sky constant buffer has also been modified since we are only translating one cloud layer. In the shader itself we start by sampling the noise texture to get the base perturb value that we will use to offset the sampling of the cloud texture. We then multiply it by the perturb scale and add the texture coordinates. This gives us the perturbed coordinate that we then sample the cloud texture from. Note that both are translated by the translation value from the constant buffer to create the scrolling clouds effect.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplane.ps

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

Texture2D cloudTexture : register(t0);

Texture2D perturbTexture : register(t1);

SamplerState SampleType;

cbuffer SkyBuffer

{

float translation;

float scale;

float brightness;

float padding;

};

//////////////

// TYPEDEFS //

//////////////

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

};

////////////////////////////////////////////////////////////////////////////////

// Pixel Shader

////////////////////////////////////////////////////////////////////////////////

float4 SkyPlanePixelShader(PixelInputType input) : SV\_TARGET

{

float4 perturbValue;

float4 cloudColor;

// Translate the texture coordinate sampling location by the translation value.

input.tex.x = input.tex.x + translation;

// Sample the texture value from the perturb texture using the translated texture coordinates.

perturbValue = perturbTexture.Sample(SampleType, input.tex);

// Multiply the perturb value by the perturb scale.

perturbValue = perturbValue \* scale;

// Add the texture coordinates as well as the translation value to get the perturbed texture coordinate sampling location.

perturbValue.xy = perturbValue.xy + input.tex.xy + translation;

// Now sample the color from the cloud texture using the perturbed sampling coordinates.

cloudColor = cloudTexture.Sample(SampleType, perturbValue.xy);

// Reduce the color cloud by the brightness value.

cloudColor = cloudColor \* brightness;

return cloudColor;

}

Skyplaneshaderclass.h

The SkyPlaneShaderClass header has been modified a bit to handle the slightly different inputs into the pixel shader. The SkyBufferType now just has a single translation value and the new perturb scale value.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneshaderclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SKYPLANESHADERCLASS\_H\_

#define \_SKYPLANESHADERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <d3dx11async.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: SkyPlaneShaderClass

////////////////////////////////////////////////////////////////////////////////

class SkyPlaneShaderClass

{

private:

struct MatrixBufferType

{

D3DXMATRIX world;

D3DXMATRIX view;

D3DXMATRIX projection;

};

struct SkyBufferType

{

float translation;

float scale;

float brightness;

float padding;

};

public:

SkyPlaneShaderClass();

SkyPlaneShaderClass(const SkyPlaneShaderClass&);

~SkyPlaneShaderClass();

bool Initialize(ID3D11Device\*, HWND);

void Shutdown();

bool Render(ID3D11DeviceContext\*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*, float, float, float);

private:

bool InitializeShader(ID3D11Device\*, HWND, WCHAR\*, WCHAR\*);

void ShutdownShader();

void OutputShaderErrorMessage(ID3D10Blob\*, HWND, WCHAR\*);

bool SetShaderParameters(ID3D11DeviceContext\*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*, float, float,

float);

void RenderShader(ID3D11DeviceContext\*, int);

private:

ID3D11VertexShader\* m\_vertexShader;

ID3D11PixelShader\* m\_pixelShader;

ID3D11InputLayout\* m\_layout;

ID3D11SamplerState\* m\_sampleState;

ID3D11Buffer\* m\_matrixBuffer;

ID3D11Buffer\* m\_skyBuffer;

};

#endif

Skyplaneshaderclass.cpp

The SkyPlaneShaderClass has been modified a bit to handle the slightly different inputs into the pixel shader.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "skyplaneshaderclass.h"

SkyPlaneShaderClass::SkyPlaneShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

m\_skyBuffer = 0;

}

SkyPlaneShaderClass::SkyPlaneShaderClass(const SkyPlaneShaderClass& other)

{

}

SkyPlaneShaderClass::~SkyPlaneShaderClass()

{

}

bool SkyPlaneShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

// Initialize the vertex and pixel shaders.

result = InitializeShader(device, hwnd, L"../Engine/skyplane.vs", L"../Engine/skyplane.ps");

if(!result)

{

return false;

}

return true;

}

void SkyPlaneShaderClass::Shutdown()

{

// Shutdown the vertex and pixel shaders as well as the related objects.

ShutdownShader();

return;

}

The inputs to the Render function have been changed. There is now just a single cloud texture and a single translation value. We also have a noise texture and a perturb scale value.

bool SkyPlaneShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView\* cloudTexture, ID3D11ShaderResourceView\* perturbTexture,

float translation, float scale, float brightness)

{

bool result;

// Set the shader parameters that it will use for rendering.

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, cloudTexture, perturbTexture, translation, scale, brightness);

if(!result)

{

return false;

}

// Now render the prepared buffers with the shader.

RenderShader(deviceContext, indexCount);

return true;

}

bool SkyPlaneShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[2];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC skyBufferDesc;

// Initialize the pointers this function will use to null.

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

// Compile the vertex shader code.

result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "SkyPlaneVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&vertexShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

// If there was nothing in the error message then it simply could not find the shader file itself.

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Compile the pixel shader code.

result = D3DX11CompileFromFile(psFilename, NULL, NULL, "SkyPlanePixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&pixelShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

// If there was nothing in the error message then it simply could not find the file itself.

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Create the vertex shader from the buffer.

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

// Create the pixel shader from the buffer.

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;

}

// Create the vertex input layout description.

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

// Get a count of the elements in the layout.

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

// Create the vertex input layout.

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;

}

// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

// Create a texture sampler state description.

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

// Create the texture sampler state.

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;

}

// Setup the description of the dynamic matrix constant buffer that is in the vertex shader.

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;

}

// Setup the description of the dynamic sky constant buffer that is in the pixel shader.

skyBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

skyBufferDesc.ByteWidth = sizeof(SkyBufferType);

skyBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

skyBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

skyBufferDesc.MiscFlags = 0;

skyBufferDesc.StructureByteStride = 0;

// Create the sky buffer pointer so we can access the pixel shader constant buffer from within this class.

result = device->CreateBuffer(&skyBufferDesc, NULL, &m\_skyBuffer);

if(FAILED(result))

{

return false;

}

return true;

}

void SkyPlaneShaderClass::ShutdownShader()

{

// Release the sky constant buffer.

if(m\_skyBuffer)

{

m\_skyBuffer->Release();

m\_skyBuffer = 0;

}

// Release the matrix constant buffer.

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;

}

// Release the sampler states.

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;

}

// Release the layout.

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;

}

// Release the pixel shader.

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;

}

// Release the vertex shader.

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;

}

return;

}

void SkyPlaneShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

// Get a pointer to the error message text buffer.

compileErrors = (char\*)(errorMessage->GetBufferPointer());

// Get the length of the message.

bufferSize = errorMessage->GetBufferSize();

// Open a file to write the error message to.

fout.open("shader-error.txt");

// Write out the error message.

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

// Close the file.

fout.close();

// Release the error message.

errorMessage->Release();

errorMessage = 0;

// Pop a message up on the screen to notify the user to check the text file for compile errors.

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;

}

The inputs to SetShaderParameters have also changed. There is now just a single cloud texture and a single translation value. We also have a noise texture and a perturb scale value.

bool SkyPlaneShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView\* cloudTexture,

ID3D11ShaderResourceView\* perturbTexture, float translation, float scale, float brightness)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

MatrixBufferType\* dataPtr;

SkyBufferType\* dataPtr2;

unsigned int bufferNumber;

// Transpose the matrices to prepare them for the shader.

D3DXMatrixTranspose(&worldMatrix, &worldMatrix);

D3DXMatrixTranspose(&viewMatrix, &viewMatrix);

D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

// Lock the constant buffer so it can be written to.

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr = (MatrixBufferType\*)mappedResource.pData;

// Copy the matrices into the constant buffer.

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

// Unlock the constant buffer.

deviceContext->Unmap(m\_matrixBuffer, 0);

// Set the position of the constant buffer in the vertex shader.

bufferNumber = 0;

// Finally set the constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

// Lock the sky constant buffer so it can be written to.

result = deviceContext->Map(m\_skyBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the sky constant buffer.

dataPtr2 = (SkyBufferType\*)mappedResource.pData;

The constant buffer inputs are set here with the new parameters.

// Copy the data into the sky constant buffer.

dataPtr2->translation = translation;

dataPtr2->scale = scale;

dataPtr2->brightness = brightness;

dataPtr2->padding = 0.0f;

// Unlock the sky constant buffer.

deviceContext->Unmap(m\_skyBuffer, 0);

// Set the position of the sky constant buffer in the pixel shader.

bufferNumber = 0;

// Now set the sky constant buffer in the pixel shader with the updated values.

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_skyBuffer);

// Set the shader texture resources in the pixel shader.

deviceContext->PSSetShaderResources(0, 1, &cloudTexture);

We set the noise texture in the pixel shader instead of the second cloud layer.

deviceContext->PSSetShaderResources(1, 1, &perturbTexture);

return true;

}

void SkyPlaneShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

// Set the vertex input layout.

deviceContext->IASetInputLayout(m\_layout);

// Set the vertex and pixel shaders that will be used to render the triangles.

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

// Set the sampler state in the pixel shader.

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

// Render the triangles.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}

Skyplaneclass.h

The SkyPlaneClass has some changes also to accommodate the new input values into the pixel shader.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SKYPLANECLASS\_H\_

#define \_SKYPLANECLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "textureclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: SkyPlaneClass

////////////////////////////////////////////////////////////////////////////////

class SkyPlaneClass

{

private:

struct SkyPlaneType

{

float x, y, z;

float tu, tv;

};

struct VertexType

{

D3DXVECTOR3 position;

D3DXVECTOR2 texture;

};

public:

SkyPlaneClass();

SkyPlaneClass(const SkyPlaneClass&);

~SkyPlaneClass();

bool Initialize(ID3D11Device\*, WCHAR\*, WCHAR\*);

void Shutdown();

void Render(ID3D11DeviceContext\*);

void Frame();

int GetIndexCount();

ID3D11ShaderResourceView\* GetCloudTexture();

We have a new function that returns the perturb/noise texture instead of the second cloud layer.

ID3D11ShaderResourceView\* GetPerturbTexture();

We also have three new functions to pass the perturb scale, brightness, and translation into the pixel shader.

float GetScale();

float GetBrightness();

float GetTranslation();

private:

bool InitializeSkyPlane(int, float, float, float, int);

void ShutdownSkyPlane();

bool InitializeBuffers(ID3D11Device\*, int);

void ShutdownBuffers();

void RenderBuffers(ID3D11DeviceContext\*);

bool LoadTextures(ID3D11Device\*, WCHAR\*, WCHAR\*);

void ReleaseTextures();

private:

SkyPlaneType\* m\_skyPlane;

int m\_vertexCount, m\_indexCount;

ID3D11Buffer \*m\_vertexBuffer, \*m\_indexBuffer;

There are new variables for the noise texture, the scale, and the single translation value.

TextureClass \*m\_CloudTexture, \*m\_PerturbTexture;

float m\_scale, m\_brightness, m\_translation;

};

#endif

Skyplaneclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "skyplaneclass.h"

SkyPlaneClass::SkyPlaneClass()

{

m\_skyPlane = 0;

m\_vertexBuffer = 0;

m\_indexBuffer = 0;

m\_CloudTexture = 0;

Initialize the new perturb texture to null in the class constructor.

m\_PerturbTexture = 0;

}

SkyPlaneClass::SkyPlaneClass(const SkyPlaneClass& other)

{

}

SkyPlaneClass::~SkyPlaneClass()

{

}

bool SkyPlaneClass::Initialize(ID3D11Device\* device, WCHAR\* cloudTextureFilename, WCHAR\* perturbTextureFilename)

{

int skyPlaneResolution, textureRepeat;

float skyPlaneWidth, skyPlaneTop, skyPlaneBottom;

bool result;

// Set the sky plane parameters.

skyPlaneResolution = 50;

skyPlaneWidth = 10.0f;

skyPlaneTop = 0.5f;

skyPlaneBottom = 0.0f;

textureRepeat = 2;

Set the perturb scale value, note that this has a drastic effect on how the perturbed sampling coordinates are selected.

// Set the sky plane shader related parameters.

m\_scale = 0.3f;

m\_brightness = 0.5f;

We are only using a single translation value in this tutorial. We initialize it to zero here.

// Initialize the translation to zero.

m\_translation = 0.0f;

// Create the sky plane.

result = InitializeSkyPlane(skyPlaneResolution, skyPlaneWidth, skyPlaneTop, skyPlaneBottom, textureRepeat);

if(!result)

{

return false;

}

// Create the vertex and index buffer for the sky plane.

result = InitializeBuffers(device, skyPlaneResolution);

if(!result)

{

return false;

}

We load the textures here and send in the perturb texture as the second texture input.

// Load the sky plane textures.

result = LoadTextures(device, cloudTextureFilename, perturbTextureFilename);

if(!result)

{

return false;

}

return true;

}

void SkyPlaneClass::Shutdown()

{

// Release the sky plane textures.

ReleaseTextures();

// Release the vertex and index buffer that were used for rendering the sky plane.

ShutdownBuffers();

// Release the sky plane array.

ShutdownSkyPlane();

return;

}

void SkyPlaneClass::Render(ID3D11DeviceContext\* deviceContext)

{

// Render the sky plane.

RenderBuffers(deviceContext);

return;

}

void SkyPlaneClass::Frame()

{

Each frame we increase the translation by a small amount. If you are going to expand on this you will want to modify the translation amount to be based on the frame time. You will also want to slow it down a bit as I made it fast for this tutorial so that the effect is very clear to see due to the high speed time lapse.

// Increment the texture translation value each frame.

m\_translation += 0.0001f;

if(m\_translation > 1.0f)

{

m\_translation -= 1.0f;

}

return;

}

int SkyPlaneClass::GetIndexCount()

{

return m\_indexCount;

}

ID3D11ShaderResourceView\* SkyPlaneClass::GetCloudTexture()

{

return m\_CloudTexture->GetTexture();

}

The new GetPerturbTexture function returns the noise texture used for perturbing the cloud sampling.

ID3D11ShaderResourceView\* SkyPlaneClass::GetPerturbTexture()

{

return m\_PerturbTexture->GetTexture();

}

GetScale return the perturb scale for the pixel shader rendering.

float SkyPlaneClass::GetScale()

{

return m\_scale;

}

float SkyPlaneClass::GetBrightness()

{

return m\_brightness;

}

The GetTranslation function returns the single translation value we used for rotating the clouds over the sky plane.

float SkyPlaneClass::GetTranslation()

{

return m\_translation;

}

bool SkyPlaneClass::InitializeSkyPlane(int skyPlaneResolution, float skyPlaneWidth, float skyPlaneTop, float skyPlaneBottom, int textureRepeat)

{

float quadSize, radius, constant, textureDelta;

int i, j, index;

float positionX, positionY, positionZ, tu, tv;

// Create the array to hold the sky plane coordinates.

m\_skyPlane = new SkyPlaneType[(skyPlaneResolution + 1) \* (skyPlaneResolution + 1)];

if(!m\_skyPlane)

{

return false;

}

// Determine the size of each quad on the sky plane.

quadSize = skyPlaneWidth / (float)skyPlaneResolution;

// Calculate the radius of the sky plane based on the width.

radius = skyPlaneWidth / 2.0f;

// Calculate the height constant to increment by.

constant = (skyPlaneTop - skyPlaneBottom) / (radius \* radius);

// Calculate the texture coordinate increment value.

textureDelta = (float)textureRepeat / (float)skyPlaneResolution;

// Loop through the sky plane and build the coordinates based on the increment values given.

for(j=0; j<=skyPlaneResolution; j++)

{

for(i=0; i<=skyPlaneResolution; i++)

{

// Calculate the vertex coordinates.

positionX = (-0.5f \* skyPlaneWidth) + ((float)i \* quadSize);

positionZ = (-0.5f \* skyPlaneWidth) + ((float)j \* quadSize);

positionY = skyPlaneTop - (constant \* ((positionX \* positionX) + (positionZ \* positionZ)));

// Calculate the texture coordinates.

tu = (float)i \* textureDelta;

tv = (float)j \* textureDelta;

// Calculate the index into the sky plane array to add this coordinate.

index = j \* (skyPlaneResolution + 1) + i;

// Add the coordinates to the sky plane array.

m\_skyPlane[index].x = positionX;

m\_skyPlane[index].y = positionY;

m\_skyPlane[index].z = positionZ;

m\_skyPlane[index].tu = tu;

m\_skyPlane[index].tv = tv;

}

}

return true;

}

void SkyPlaneClass::ShutdownSkyPlane()

{

// Release the sky plane array.

if(m\_skyPlane)

{

delete [] m\_skyPlane;

m\_skyPlane = 0;

}

return;

}

bool SkyPlaneClass::InitializeBuffers(ID3D11Device\* device, int skyPlaneResolution)

{

VertexType\* vertices;

unsigned long\* indices;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

int i, j, index, index1, index2, index3, index4;

// Calculate the number of vertices in the sky plane mesh.

m\_vertexCount = (skyPlaneResolution + 1) \* (skyPlaneResolution + 1) \* 6;

// Set the index count to the same as the vertex count.

m\_indexCount = m\_vertexCount;

// Create the vertex array.

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;

}

// Create the index array.

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;

}

// Initialize the index into the vertex array.

index = 0;

// Load the vertex and index array with the sky plane array data.

for(j=0; j<skyPlaneResolution; j++)

{

for(i=0; i<skyPlaneResolution; i++)

{

index1 = j \* (skyPlaneResolution + 1) + i;

index2 = j \* (skyPlaneResolution + 1) + (i+1);

index3 = (j+1) \* (skyPlaneResolution + 1) + i;

index4 = (j+1) \* (skyPlaneResolution + 1) + (i+1);

// Triangle 1 - Upper Left

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index1].x, m\_skyPlane[index1].y, m\_skyPlane[index1].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index1].tu, m\_skyPlane[index1].tv);

indices[index] = index;

index++;

// Triangle 1 - Upper Right

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index2].x, m\_skyPlane[index2].y, m\_skyPlane[index2].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index2].tu, m\_skyPlane[index2].tv);

indices[index] = index;

index++;

// Triangle 1 - Bottom Left

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index3].x, m\_skyPlane[index3].y, m\_skyPlane[index3].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index3].tu, m\_skyPlane[index3].tv);

indices[index] = index;

index++;

// Triangle 2 - Bottom Left

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index3].x, m\_skyPlane[index3].y, m\_skyPlane[index3].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index3].tu, m\_skyPlane[index3].tv);

indices[index] = index;

index++;

// Triangle 2 - Upper Right

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index2].x, m\_skyPlane[index2].y, m\_skyPlane[index2].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index2].tu, m\_skyPlane[index2].tv);

indices[index] = index;

index++;

// Triangle 2 - Bottom Right

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index4].x, m\_skyPlane[index4].y, m\_skyPlane[index4].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index4].tu, m\_skyPlane[index4].tv);

indices[index] = index;

index++;

}

}

// Set up the description of the vertex buffer.

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the vertex data.

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

// Now finally create the vertex buffer.

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{

return false;

}

// Set up the description of the index buffer.

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the index data.

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

// Create the index buffer.

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{

return false;

}

// Release the arrays now that the vertex and index buffers have been created and loaded.

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;

}

void SkyPlaneClass::ShutdownBuffers()

{

// Release the index buffer.

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;

}

// Release the vertex buffer.

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;

}

return;

}

void SkyPlaneClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

// Set vertex buffer stride and offset.

stride = sizeof(VertexType);

offset = 0;

// Set the vertex buffer to active in the input assembler so it can be rendered.

deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

// Set the index buffer to active in the input assembler so it can be rendered.

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}

bool SkyPlaneClass::LoadTextures(ID3D11Device\* device, WCHAR\* textureFilename1, WCHAR\* textureFilename2)

{

bool result;

// Create the cloud texture object.

m\_CloudTexture = new TextureClass;

if(!m\_CloudTexture)

{

return false;

}

// Initialize the cloud texture object.

result = m\_CloudTexture->Initialize(device, textureFilename1);

if(!result)

{

return false;

}

We load the new noise texture here.

// Create the perturb texture object.

m\_PerturbTexture = new TextureClass;

if(!m\_PerturbTexture)

{

return false;

}

// Initialize the perturb texture object.

result = m\_PerturbTexture->Initialize(device, textureFilename2);

if(!result)

{

return false;

}

return true;

}

void SkyPlaneClass::ReleaseTextures()

{

The new noise texture is released here in the ReleaseTextures function.

// Release the texture objects.

if(m\_PerturbTexture)

{

m\_PerturbTexture->Shutdown();

delete m\_PerturbTexture;

m\_PerturbTexture = 0;

}

if(m\_CloudTexture)

{

m\_CloudTexture->Shutdown();

delete m\_CloudTexture;

m\_CloudTexture = 0;

}

return;

}

Applicationclass.h

The ApplicationClass header has not changed for this tutorial.

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_APPLICATIONCLASS\_H\_

#define \_APPLICATIONCLASS\_H\_

/////////////

// GLOBALS //

/////////////

const bool FULL\_SCREEN = true;

const bool VSYNC\_ENABLED = true;

const float SCREEN\_DEPTH = 1000.0f;

const float SCREEN\_NEAR = 0.1f;

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "inputclass.h"

#include "d3dclass.h"

#include "cameraclass.h"

#include "terrainclass.h"

#include "timerclass.h"

#include "positionclass.h"

#include "fpsclass.h"

#include "cpuclass.h"

#include "fontshaderclass.h"

#include "textclass.h"

#include "terrainshaderclass.h"

#include "lightclass.h"

#include "skydomeclass.h"

#include "skydomeshaderclass.h"

#include "skyplaneclass.h"

#include "skyplaneshaderclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: ApplicationClass

////////////////////////////////////////////////////////////////////////////////

class ApplicationClass

{

public:

ApplicationClass();

ApplicationClass(const ApplicationClass&);

~ApplicationClass();

bool Initialize(HINSTANCE, HWND, int, int);

void Shutdown();

bool Frame();

private:

bool HandleInput(float);

bool RenderGraphics();

private:

InputClass\* m\_Input;

D3DClass\* m\_Direct3D;

CameraClass\* m\_Camera;

TerrainClass\* m\_Terrain;

TimerClass\* m\_Timer;

PositionClass\* m\_Position;

FpsClass\* m\_Fps;

CpuClass\* m\_Cpu;

FontShaderClass\* m\_FontShader;

TextClass\* m\_Text;

TerrainShaderClass\* m\_TerrainShader;

LightClass\* m\_Light;

SkyDomeClass\* m\_SkyDome;

SkyDomeShaderClass\* m\_SkyDomeShader;

SkyPlaneClass \*m\_SkyPlane;

SkyPlaneShaderClass\* m\_SkyPlaneShader;

};

#endif

Applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "applicationclass.h"

ApplicationClass::ApplicationClass()

{

m\_Input = 0;

m\_Direct3D = 0;

m\_Camera = 0;

m\_Terrain = 0;

m\_Timer = 0;

m\_Position = 0;

m\_Fps = 0;

m\_Cpu = 0;

m\_FontShader = 0;

m\_Text = 0;

m\_TerrainShader = 0;

m\_Light = 0;

m\_SkyDome = 0;

m\_SkyDomeShader = 0;

m\_SkyPlane = 0;

m\_SkyPlaneShader = 0;

}

ApplicationClass::ApplicationClass(const ApplicationClass& other)

{

}

ApplicationClass::~ApplicationClass()

{

}

bool ApplicationClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)

{

bool result;

float cameraX, cameraY, cameraZ;

D3DXMATRIX baseViewMatrix;

char videoCard[128];

int videoMemory;

// Create the input object. The input object will be used to handle reading the keyboard and mouse input from the user.

m\_Input = new InputClass;

if(!m\_Input)

{

return false;

}

// Initialize the input object.

result = m\_Input->Initialize(hinstance, hwnd, screenWidth, screenHeight);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the input object.", L"Error", MB\_OK);

return false;

}

// Create the Direct3D object.

m\_Direct3D = new D3DClass;

if(!m\_Direct3D)

{

return false;

}

// Initialize the Direct3D object.

result = m\_Direct3D->Initialize(screenWidth, screenHeight, VSYNC\_ENABLED, hwnd, FULL\_SCREEN, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize DirectX 11.", L"Error", MB\_OK);

return false;

}

// Create the camera object.

m\_Camera = new CameraClass;

if(!m\_Camera)

{

return false;

}

// Initialize a base view matrix with the camera for 2D user interface rendering.

m\_Camera->SetPosition(0.0f, 0.0f, -1.0f);

m\_Camera->Render();

m\_Camera->GetViewMatrix(baseViewMatrix);

// Set the initial position of the camera.

cameraX = 50.0f;

cameraY = 2.0f;

cameraZ = -7.0f;

m\_Camera->SetPosition(cameraX, cameraY, cameraZ);

// Create the terrain object.

m\_Terrain = new TerrainClass;

if(!m\_Terrain)

{

return false;

}

// Initialize the terrain object.

result = m\_Terrain->Initialize(m\_Direct3D->GetDevice(), "../Engine/data/heightmap01.bmp", L"../Engine/data/dirt01.dds", "../Engine/data/colorm01.bmp");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain object.", L"Error", MB\_OK);

return false;

}

// Create the timer object.

m\_Timer = new TimerClass;

if(!m\_Timer)

{

return false;

}

// Initialize the timer object.

result = m\_Timer->Initialize();

if(!result)

{

MessageBox(hwnd, L"Could not initialize the timer object.", L"Error", MB\_OK);

return false;

}

// Create the position object.

m\_Position = new PositionClass;

if(!m\_Position)

{

return false;

}

// Set the initial position of the viewer to the same as the initial camera position.

m\_Position->SetPosition(cameraX, cameraY, cameraZ);

// Create the fps object.

m\_Fps = new FpsClass;

if(!m\_Fps)

{

return false;

}

// Initialize the fps object.

m\_Fps->Initialize();

// Create the cpu object.

m\_Cpu = new CpuClass;

if(!m\_Cpu)

{

return false;

}

// Initialize the cpu object.

m\_Cpu->Initialize();

// Create the font shader object.

m\_FontShader = new FontShaderClass;

if(!m\_FontShader)

{

return false;

}

// Initialize the font shader object.

result = m\_FontShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the font shader object.", L"Error", MB\_OK);

return false;

}

// Create the text object.

m\_Text = new TextClass;

if(!m\_Text)

{

return false;

}

// Initialize the text object.

result = m\_Text->Initialize(m\_Direct3D->GetDevice(), m\_Direct3D->GetDeviceContext(), hwnd, screenWidth, screenHeight, baseViewMatrix);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the text object.", L"Error", MB\_OK);

return false;

}

// Retrieve the video card information.

m\_Direct3D->GetVideoCardInfo(videoCard, videoMemory);

// Set the video card information in the text object.

result = m\_Text->SetVideoCardInfo(videoCard, videoMemory, m\_Direct3D->GetDeviceContext());

if(!result)

{

MessageBox(hwnd, L"Could not set video card info in the text object.", L"Error", MB\_OK);

return false;

}

// Create the terrain shader object.

m\_TerrainShader = new TerrainShaderClass;

if(!m\_TerrainShader)

{

return false;

}

// Initialize the terrain shader object.

result = m\_TerrainShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain shader object.", L"Error", MB\_OK);

return false;

}

// Create the light object.

m\_Light = new LightClass;

if(!m\_Light)

{

return false;

}

// Initialize the light object.

m\_Light->SetAmbientColor(0.05f, 0.05f, 0.05f, 1.0f);

m\_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);

m\_Light->SetDirection(-0.5f, -1.0f, 0.0f);

// Create the sky dome object.

m\_SkyDome = new SkyDomeClass;

if(!m\_SkyDome)

{

return false;

}

// Initialize the sky dome object.

result = m\_SkyDome->Initialize(m\_Direct3D->GetDevice());

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome object.", L"Error", MB\_OK);

return false;

}

// Create the sky dome shader object.

m\_SkyDomeShader = new SkyDomeShaderClass;

if(!m\_SkyDomeShader)

{

return false;

}

// Initialize the sky dome shader object.

result = m\_SkyDomeShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome shader object.", L"Error", MB\_OK);

return false;

}

// Create the sky plane object.

m\_SkyPlane = new SkyPlaneClass;

if(!m\_SkyPlane)

{

return false;

}

The perturb noise texture is now sent in as the second texture input.

// Initialize the sky plane object.

result = m\_SkyPlane->Initialize(m\_Direct3D->GetDevice(), L"../Engine/data/cloud001.dds", L"../Engine/data/perturb001.dds");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky plane object.", L"Error", MB\_OK);

return false;

}

// Create the sky plane shader object.

m\_SkyPlaneShader = new SkyPlaneShaderClass;

if(!m\_SkyPlaneShader)

{

return false;

}

// Initialize the sky plane shader object.

result = m\_SkyPlaneShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky plane shader object.", L"Error", MB\_OK);

return false;

}

return true;

}

void ApplicationClass::Shutdown()

{

// Release the sky plane shader object.

if(m\_SkyPlaneShader)

{

m\_SkyPlaneShader->Shutdown();

delete m\_SkyPlaneShader;

m\_SkyPlaneShader = 0;

}

// Release the sky plane object.

if(m\_SkyPlane)

{

m\_SkyPlane->Shutdown();

delete m\_SkyPlane;

m\_SkyPlane = 0;

}

// Release the sky dome shader object.

if(m\_SkyDomeShader)

{

m\_SkyDomeShader->Shutdown();

delete m\_SkyDomeShader;

m\_SkyDomeShader = 0;

}

// Release the sky dome object.

if(m\_SkyDome)

{

m\_SkyDome->Shutdown();

delete m\_SkyDome;

m\_SkyDome = 0;

}

// Release the light object.

if(m\_Light)

{

delete m\_Light;

m\_Light = 0;

}

// Release the terrain shader object.

if(m\_TerrainShader)

{

m\_TerrainShader->Shutdown();

delete m\_TerrainShader;

m\_TerrainShader = 0;

}

// Release the text object.

if(m\_Text)

{

m\_Text->Shutdown();

delete m\_Text;

m\_Text = 0;

}

// Release the font shader object.

if(m\_FontShader)

{

m\_FontShader->Shutdown();

delete m\_FontShader;

m\_FontShader = 0;

}

// Release the cpu object.

if(m\_Cpu)

{

m\_Cpu->Shutdown();

delete m\_Cpu;

m\_Cpu = 0;

}

// Release the fps object.

if(m\_Fps)

{

delete m\_Fps;

m\_Fps = 0;

}

// Release the position object.

if(m\_Position)

{

delete m\_Position;

m\_Position = 0;

}

// Release the timer object.

if(m\_Timer)

{

delete m\_Timer;

m\_Timer = 0;

}

// Release the terrain object.

if(m\_Terrain)

{

m\_Terrain->Shutdown();

delete m\_Terrain;

m\_Terrain = 0;

}

// Release the camera object.

if(m\_Camera)

{

delete m\_Camera;

m\_Camera = 0;

}

// Release the Direct3D object.

if(m\_Direct3D)

{

m\_Direct3D->Shutdown();

delete m\_Direct3D;

m\_Direct3D = 0;

}

// Release the input object.

if(m\_Input)

{

m\_Input->Shutdown();

delete m\_Input;

m\_Input = 0;

}

return;

}

bool ApplicationClass::Frame()

{

bool result;

// Read the user input.

result = m\_Input->Frame();

if(!result)

{

return false;

}

// Check if the user pressed escape and wants to exit the application.

if(m\_Input->IsEscapePressed() == true)

{

return false;

}

// Update the system stats.

m\_Timer->Frame();

m\_Fps->Frame();

m\_Cpu->Frame();

// Update the FPS value in the text object.

result = m\_Text->SetFps(m\_Fps->GetFps(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the CPU usage value in the text object.

result = m\_Text->SetCpu(m\_Cpu->GetCpuPercentage(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Do the frame input processing.

result = HandleInput(m\_Timer->GetTime());

if(!result)

{

return false;

}

// Do the sky plane frame processing.

m\_SkyPlane->Frame();

// Render the graphics.

result = RenderGraphics();

if(!result)

{

return false;

}

return result;

}

bool ApplicationClass::HandleInput(float frameTime)

{

bool keyDown, result;

float posX, posY, posZ, rotX, rotY, rotZ;

// Set the frame time for calculating the updated position.

m\_Position->SetFrameTime(frameTime);

// Handle the input.

keyDown = m\_Input->IsLeftPressed();

m\_Position->TurnLeft(keyDown);

keyDown = m\_Input->IsRightPressed();

m\_Position->TurnRight(keyDown);

keyDown = m\_Input->IsUpPressed();

m\_Position->MoveForward(keyDown);

keyDown = m\_Input->IsDownPressed();

m\_Position->MoveBackward(keyDown);

keyDown = m\_Input->IsAPressed();

m\_Position->MoveUpward(keyDown);

keyDown = m\_Input->IsZPressed();

m\_Position->MoveDownward(keyDown);

keyDown = m\_Input->IsPgUpPressed();

m\_Position->LookUpward(keyDown);

keyDown = m\_Input->IsPgDownPressed();

m\_Position->LookDownward(keyDown);

// Get the view point position/rotation.

m\_Position->GetPosition(posX, posY, posZ);

m\_Position->GetRotation(rotX, rotY, rotZ);

// Set the position of the camera.

m\_Camera->SetPosition(posX, posY, posZ);

m\_Camera->SetRotation(rotX, rotY, rotZ);

// Update the position values in the text object.

result = m\_Text->SetCameraPosition(posX, posY, posZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the rotation values in the text object.

result = m\_Text->SetCameraRotation(rotX, rotY, rotZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

return true;

}

bool ApplicationClass::RenderGraphics()

{

D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix;

D3DXVECTOR3 cameraPosition;

bool result;

// Clear the scene.

m\_Direct3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

// Generate the view matrix based on the camera's position.

m\_Camera->Render();

// Get the world, view, projection, and ortho matrices from the camera and Direct3D objects.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

m\_Direct3D->GetOrthoMatrix(orthoMatrix);

// Get the position of the camera.

cameraPosition = m\_Camera->GetPosition();

// Translate the sky dome to be centered around the camera position.

D3DXMatrixTranslation(&worldMatrix, cameraPosition.x, cameraPosition.y, cameraPosition.z);

// Turn off back face culling.

m\_Direct3D->TurnOffCulling();

// Turn off the Z buffer.

m\_Direct3D->TurnZBufferOff();

// Render the sky dome using the sky dome shader.

m\_SkyDome->Render(m\_Direct3D->GetDeviceContext());

m\_SkyDomeShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyDome->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyDome->GetApexColor(), m\_SkyDome->GetCenterColor());

// Turn back face culling back on.

m\_Direct3D->TurnOnCulling();

// Enable additive blending so the clouds blend with the sky dome color.

m\_Direct3D->EnableSecondBlendState();

// Render the sky plane using the sky plane shader.

m\_SkyPlane->Render(m\_Direct3D->GetDeviceContext());

The sky plane is now rendered using the new inputs for the pixel shader.

m\_SkyPlaneShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyPlane->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyPlane->GetCloudTexture(), m\_SkyPlane->GetPerturbTexture(), m\_SkyPlane->GetTranslation(), m\_SkyPlane->GetScale(),

m\_SkyPlane->GetBrightness());

// Turn off blending.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on.

m\_Direct3D->TurnZBufferOn();

// Reset the world matrix.

m\_Direct3D->GetWorldMatrix(worldMatrix);

// Render the terrain using the terrain shader.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

result = m\_TerrainShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_Light->GetAmbientColor(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(), m\_Terrain->GetTexture());

if(!result)

{

return false;

}

// Turn off the Z buffer to begin all 2D rendering.

m\_Direct3D->TurnZBufferOff();

// Turn on the alpha blending before rendering the text.

m\_Direct3D->TurnOnAlphaBlending();

// Render the text user interface elements.

result = m\_Text->Render(m\_Direct3D->GetDeviceContext(), m\_FontShader, worldMatrix, orthoMatrix);

if(!result)

{

return false;

}

// Turn off alpha blending after rendering the text.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on now that all 2D rendering has completed.

m\_Direct3D->TurnZBufferOn();

// Present the rendered scene to the screen.

m\_Direct3D->EndScene();

return true;

}

Summary

We now have clouds that transform as they move across the sky plane. I did speed up the translation so the effect is very clear, but you will want to slow it down for a more realistic effect.



To Do Exercises

1. Compile and run the program. Use the PgUp keys to look up into the sky to see the effect. Press escape to quit when done.

2. Modify the scale value in the SkyPlane class to see the effect it has on the pixel shader.

3. Modify the noise texture to see its effect on the sampling.

4. Add a second layer of perturbed clouds, make sure they have their own translation and perturb scale also. Combine the layers just like the previous tutorial.

5. Use a higher resolution texture for the clouds and the noise to see the improved look you get.

Source Code

Source Code and Data Files: [tersrc12.zip](http://web.archive.org/web/20140722210824/http:/rastertek.com/tersrc12.zip)

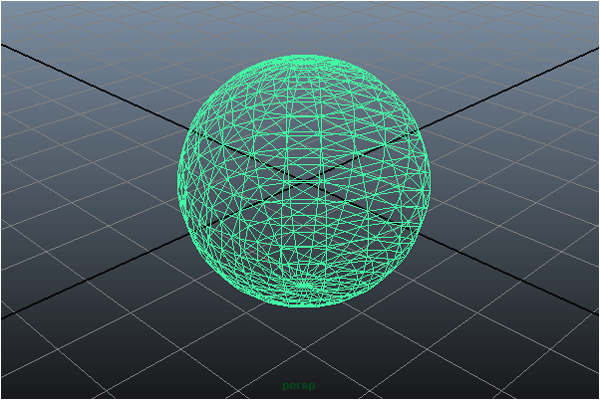
Executable: [terexe12.zip](http://web.archive.org/web/20140722210824/http:/rastertek.com/terexe12.zip)

**Tutorial 10: Sky Domes**

http://web.archive.org/web/20140722202020/http:/rastertek.com/pic1001.gif

This DirectX 11 terrain tutorial will cover how to implement sky domes so that we can generate a gradient colored sky that works well as a background to our terrain. The code in this tutorial builds off the color mapped terrain tutorial code.

To implement a sky dome you first need a sky dome model. The easiest one to use is just a regular sphere. For this tutorial I went into Maya and created a sphere with a radius of 2.0 and 20 subdivisions. I then triangulated it and exported it as a .obj formatted model. In the DirectX 11 tutorial section I cover loading and rendering basic .obj models from Maya so if you haven't seen those tutorials you may want to review them before proceeding. Also note that the radius of 2.0 is very important as the pixel shader is going to be dependent on that size value.



The next step to using a sky dome is that we need to render the sky dome around the camera position at all times. This way the sky dome is always surrounding the viewer regardless of where they move to. We do this by getting the camera position each frame and start the frame by translating the sky dome to be centered at the camera position and then render it there.

The third step is that we need to turn off back face culling when rendering the sky dome. Since we are inside the sky dome at all times the graphics card would cull the polygons since they are facing the wrong direction. So we turn off back face culling using a different raster state, render the sky dome, and then turn back face culling back on by re-enabling the original raster state.

The fourth step is that we also need to disable the Z buffer before rendering. If we don't turn it off we can't see anything outside the sky dome, only what is inside. Turning off the Z buffer before we render allows us to draw the sky dome entirely to the back buffer overwriting everything else regardless of distance. This is also the reason we need to render the sky dome before rendering the terrain or anything else. Once the sky dome is rendered we turn the Z buffer back on so that everything else is rendered according to depth again.

The fifth and final step is implemented in the shader. In the pixel shader we will pass in the current position of the pixel that we are rendering for the sky dome. We will treat the Y coordinate of this position as the height. And since the radius of the sphere was 2.0 this will be translated as +1.0f as the top of the sky dome and -1.0f as the bottom of the sky dome. And since we know the height is between +1.0 and -1.0 we can color the sky dome using the height as the interpolation value between two different colors to create a sky color gradient.

Now for the code section of this tutorial I have added two new classes called SkyDomeClass and SkyDomeShaderClass. I have also added the skydome vertex and pixel shader HLSL programs. The SkyDomeClass basically encapsulates the sphere model of the sky dome as well as the two colors for the gradient. The SkyDomeShaderClass is used for rendering the sky dome model.

We will start the code section now by examining the SkyDomeClass.

Skydomeclass.h

The SkyDomeClass is basically the ModelClass from the DirectX 11 tutorials re-written for the purposes of rendering a sky dome.

////////////////////////////////////////////////////////////////////////////////

// Filename: skydomeclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SKYDOMECLASS\_H\_

#define \_SKYDOMECLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: SkyDomeClass

////////////////////////////////////////////////////////////////////////////////

class SkyDomeClass

{

private:

struct ModelType

{

float x, y, z;

float tu, tv;

float nx, ny, nz;

};

The VertexType for the sky dome only requires position. We generate the colors based on the height of the sky dome.

struct VertexType

{

D3DXVECTOR3 position;

};

public:

SkyDomeClass();

SkyDomeClass(const SkyDomeClass&);

~SkyDomeClass();

bool Initialize(ID3D11Device\*);

void Shutdown();

void Render(ID3D11DeviceContext\*);

int GetIndexCount();

GetApexColor and GetCenterColor are the two helper functions that retrieve the two colors for the sky dome gradient during rendering.

D3DXVECTOR4 GetApexColor();

D3DXVECTOR4 GetCenterColor();

private:

bool LoadSkyDomeModel(char\*);

void ReleaseSkyDomeModel();

bool InitializeBuffers(ID3D11Device\*);

void ReleaseBuffers();

void RenderBuffers(ID3D11DeviceContext\*);

private:

ModelType\* m\_model;

int m\_vertexCount, m\_indexCount;

ID3D11Buffer \*m\_vertexBuffer, \*m\_indexBuffer;

D3DXVECTOR4 m\_apexColor, m\_centerColor;

};

#endif

Skydomeclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: skydomeclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "skydomeclass.h"

The class constructor initializes the private member pointers to null.

SkyDomeClass::SkyDomeClass()

{

m\_model = 0;

m\_vertexBuffer = 0;

m\_indexBuffer = 0;

}

SkyDomeClass::SkyDomeClass(const SkyDomeClass& other)

{

}

SkyDomeClass::~SkyDomeClass()

{

}

The Initialize function first loads the sky dome model into the m\_model structure. After that InitializeBuffers is called which loads the sky dome model into a vertex and index buffer that can be rendered by the video card. And finally we set the two colors of the sky dome. The apex color is the color at the top of the sky dome. The center color is the color at the horizon of the sky dome. In this tutorial everything below the horizon is set the be the same color as the horizon. So basically the gradient only goes from the top of the sky dome to the horizon.

bool SkyDomeClass::Initialize(ID3D11Device\* device)

{

bool result;

// Load in the sky dome model.

result = LoadSkyDomeModel("../Engine/data/skydome.txt");

if(!result)

{

return false;

}

// Load the sky dome into a vertex and index buffer for rendering.

result = InitializeBuffers(device);

if(!result)

{

return false;

}

// Set the color at the top of the sky dome.

m\_apexColor = D3DXVECTOR4(0.0f, 0.15f, 0.66f, 1.0f);

// Set the color at the center of the sky dome.

m\_centerColor = D3DXVECTOR4(0.81f, 0.38f, 0.66f, 1.0f);

return true;

}

The Shutdown function releases the sky dome model and the vertex and index buffers.

void SkyDomeClass::Shutdown()

{

// Release the vertex and index buffer that were used for rendering the sky dome.

ReleaseBuffers();

// Release the sky dome model.

ReleaseSkyDomeModel();

return;

}

The Render function calls the RenderBuffers function to put the sky dome geometry on the graphics pipeline for rendering.

void SkyDomeClass::Render(ID3D11DeviceContext\* deviceContext)

{

// Render the sky dome.

RenderBuffers(deviceContext);

return;

}

GetIndexCount returns the index count and is used for rendering the sky dome.

int SkyDomeClass::GetIndexCount()

{

return m\_indexCount;

}

GetApexColor returns the color of the sky dome at the very top.

D3DXVECTOR4 SkyDomeClass::GetApexColor()

{

return m\_apexColor;

}

GetCenterColor returns the color of the sky dome at the horizon (or 0.0f to be exact).

D3DXVECTOR4 SkyDomeClass::GetCenterColor()

{

return m\_centerColor;

}

The LoadSkyDomeModel function loads in the sky dome model from our file format which was created by converting the .obj formatted sphere model.

bool SkyDomeClass::LoadSkyDomeModel(char\* filename)

{

ifstream fin;

char input;

int i;

// Open the model file.

fin.open(filename);

// If it could not open the file then exit.

if(fin.fail())

{

return false;

}

// Read up to the value of vertex count.

fin.get(input);

while(input != ':')

{

fin.get(input);

}

// Read in the vertex count.

fin >> m\_vertexCount;

// Set the number of indices to be the same as the vertex count.

m\_indexCount = m\_vertexCount;

// Create the model using the vertex count that was read in.

m\_model = new ModelType[m\_vertexCount];

if(!m\_model)

{

return false;

}

// Read up to the beginning of the data.

fin.get(input);

while(input != ':')

{

fin.get(input);

}

fin.get(input);

fin.get(input);

// Read in the vertex data.

for(i=0; i<m\_vertexCount; i++)

{

fin >> m\_model[i].x >> m\_model[i].y >> m\_model[i].z;

fin >> m\_model[i].tu >> m\_model[i].tv;

fin >> m\_model[i].nx >> m\_model[i].ny >> m\_model[i].nz;

}

// Close the model file.

fin.close();

return true;

}

The ReleaseSkyDomeModel function releases the sky dome model structure.

void SkyDomeClass::ReleaseSkyDomeModel()

{

if(m\_model)

{

delete [] m\_model;

m\_model = 0;

}

return;

}

The InitializeBuffers function loads the sky dome model structure into the vertex and index buffer.

bool SkyDomeClass::InitializeBuffers(ID3D11Device\* device)

{

VertexType\* vertices;

unsigned long\* indices;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

int i;

// Create the vertex array.

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;

}

// Create the index array.

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;

}

// Load the vertex array and index array with data.

for(i=0; i<m\_vertexCount; i++)

{

vertices[i].position = D3DXVECTOR3(m\_model[i].x, m\_model[i].y, m\_model[i].z);

indices[i] = i;

}

// Set up the description of the vertex buffer.

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the vertex data.

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

// Now finally create the vertex buffer.

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{

return false;

}

// Set up the description of the index buffer.

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the index data.

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

// Create the index buffer.

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{

return false;

}

// Release the arrays now that the vertex and index buffers have been created and loaded.

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;

}

The ReleaseBuffers function releases the vertex and index buffer that were used to render the sky dome.

void SkyDomeClass::ReleaseBuffers()

{

// Release the index buffer.

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;

}

// Release the vertex buffer.

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;

}

return;

}

RenderBuffers puts the sky dome geometry on the graphics pipe line for rendering.

void SkyDomeClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

// Set vertex buffer stride and offset.

stride = sizeof(VertexType);

offset = 0;

// Set the vertex buffer to active in the input assembler so it can be rendered.

deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

// Set the index buffer to active in the input assembler so it can be rendered.

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}

Skydome.vs

The sky dome vertex shader program is very simple. We send through the regular position as usual, however we also send the position through to the pixel shader unmodified in a second variable called domePosition.

////////////////////////////////////////////////////////////////////////////////

// Filename: skydome.vs

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer MatrixBuffer

{

matrix worldMatrix;

matrix viewMatrix;

matrix projectionMatrix;

};

//////////////

// TYPEDEFS //

//////////////

struct VertexInputType

{

float4 position : POSITION;

};

struct PixelInputType

{

float4 position : SV\_POSITION;

float4 domePosition : TEXCOORD0;

};

////////////////////////////////////////////////////////////////////////////////

// Vertex Shader

////////////////////////////////////////////////////////////////////////////////

PixelInputType SkyDomeVertexShader(VertexInputType input)

{

PixelInputType output;

// Change the position vector to be 4 units for proper matrix calculations.

input.position.w = 1.0f;

// Calculate the position of the vertex against the world, view, and projection matrices.

output.position = mul(input.position, worldMatrix);

output.position = mul(output.position, viewMatrix);

output.position = mul(output.position, projectionMatrix);

// Send the unmodified position through to the pixel shader.

output.domePosition = input.position;

return output;

}

Skydome.ps

The pixel shader is where we do all the real work of rendering the sky dome. First the GradientBuffer will have had the apex and center color set so that we have the two colors to create the gradient from. In the pixel shader function we will take the height of the current pixel to determine where it is on the sky dome. We use that height as the interpolating value and then do an interpolation between the apex and the center color. The higher the height value the more the apex color will be present. The lower the height value the more the center color will be present.

Also remember that the radius of the sphere was 2.0 which gives us the -1.0f to +1.0f range in the pixel shader. If you use a different radius then you should change the values here or send them through in their own constant buffer.

////////////////////////////////////////////////////////////////////////////////

// Filename: skydome.ps

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer GradientBuffer

{

float4 apexColor;

float4 centerColor;

};

//////////////

// TYPEDEFS //

//////////////

struct PixelInputType

{

float4 position : SV\_POSITION;

float4 domePosition : TEXCOORD0;

};

////////////////////////////////////////////////////////////////////////////////

// Pixel Shader

////////////////////////////////////////////////////////////////////////////////

float4 SkyDomePixelShader(PixelInputType input) : SV\_TARGET

{

float height;

float4 outputColor;

// Determine the position on the sky dome where this pixel is located.

height = input.domePosition.y;

// The value ranges from -1.0f to +1.0f so change it to only positive values.

if(height < 0.0)

{

height = 0.0f;

}

// Determine the gradient color by interpolating between the apex and center based on the height of the pixel in the sky dome.

outputColor = lerp(centerColor, apexColor, height);

return outputColor;

}

Skydomeshaderclass.h

The SkyDomeShaderClass is used for rendering the sky dome.

////////////////////////////////////////////////////////////////////////////////

// Filename: skydomeshaderclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SKYDOMESHADERCLASS\_H\_

#define \_SKYDOMESHADERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <d3dx11async.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: SkyDomeShaderClass

////////////////////////////////////////////////////////////////////////////////

class SkyDomeShaderClass

{

private:

struct MatrixBufferType

{

D3DXMATRIX world;

D3DXMATRIX view;

D3DXMATRIX projection;

};

We need a structure for the gradient constant buffer that is located in the pixel shader.

struct GradientBufferType

{

D3DXVECTOR4 apexColor;

D3DXVECTOR4 centerColor;

};

public:

SkyDomeShaderClass();

SkyDomeShaderClass(const SkyDomeShaderClass&);

~SkyDomeShaderClass();

bool Initialize(ID3D11Device\*, HWND);

void Shutdown();

bool Render(ID3D11DeviceContext\*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, D3DXVECTOR4, D3DXVECTOR4);

private:

bool InitializeShader(ID3D11Device\*, HWND, WCHAR\*, WCHAR\*);

void ShutdownShader();

void OutputShaderErrorMessage(ID3D10Blob\*, HWND, WCHAR\*);

bool SetShaderParameters(ID3D11DeviceContext\*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, D3DXVECTOR4, D3DXVECTOR4);

void RenderShader(ID3D11DeviceContext\*, int);

private:

ID3D11VertexShader\* m\_vertexShader;

ID3D11PixelShader\* m\_pixelShader;

ID3D11InputLayout\* m\_layout;

ID3D11Buffer\* m\_matrixBuffer;

We also require a buffer variable so that the gradient colors can be set in the pixel shader.

ID3D11Buffer\* m\_gradientBuffer;

};

#endif

Skydomeshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: skydomeshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "skydomeshaderclass.h"

SkyDomeShaderClass::SkyDomeShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_matrixBuffer = 0;

m\_gradientBuffer = 0;

}

SkyDomeShaderClass::SkyDomeShaderClass(const SkyDomeShaderClass& other)

{

}

SkyDomeShaderClass::~SkyDomeShaderClass()

{

}

bool SkyDomeShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

Load in the sky dome vertex and pixel shader programs.

// Initialize the vertex and pixel shaders.

result = InitializeShader(device, hwnd, L"../Engine/skydome.vs", L"../Engine/skydome.ps");

if(!result)

{

return false;

}

return true;

}

void SkyDomeShaderClass::Shutdown()

{

// Shutdown the vertex and pixel shaders as well as the related objects.

ShutdownShader();

return;

}

bool SkyDomeShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, D3DXVECTOR4 apexColor, D3DXVECTOR4 centerColor)

{

bool result;

// Set the shader parameters that it will use for rendering.

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, apexColor, centerColor);

if(!result)

{

return false;

}

// Now render the prepared buffers with the shader.

RenderShader(deviceContext, indexCount);

return true;

}

bool SkyDomeShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[1];

unsigned int numElements;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC gradientBufferDesc;

// Initialize the pointers this function will use to null.

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

Compile and create the sky dome vertex and pixel shader programs.

// Compile the vertex shader code.

result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "SkyDomeVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&vertexShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

// If there was nothing in the error message then it simply could not find the shader file itself.

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Compile the pixel shader code.

result = D3DX11CompileFromFile(psFilename, NULL, NULL, "SkyDomePixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&pixelShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

// If there was nothing in the error message then it simply could not find the file itself.

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Create the vertex shader from the buffer.

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

// Create the pixel shader from the buffer.

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;

}

The layout only requires a single element which is the position.

// Create the vertex input layout description.

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

// Get a count of the elements in the layout.

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

// Create the vertex input layout.

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;

}

// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

// Setup the description of the dynamic matrix constant buffer that is in the vertex shader.

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;

}

We setup the gradient constant buffer here.

// Setup the description of the gradient constant buffer that is in the pixel shader.

gradientBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

gradientBufferDesc.ByteWidth = sizeof(GradientBufferType);

gradientBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

gradientBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

gradientBufferDesc.MiscFlags = 0;

gradientBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the pixel shader constant buffer from within this class.

result = device->CreateBuffer(&gradientBufferDesc, NULL, &m\_gradientBuffer);

if(FAILED(result))

{

return false;

}

return true;

}

void SkyDomeShaderClass::ShutdownShader()

{

// Release the gradient constant buffer.

if(m\_gradientBuffer)

{

m\_gradientBuffer->Release();

m\_gradientBuffer = 0;

}

// Release the matrix constant buffer.

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;

}

// Release the layout.

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;

}

// Release the pixel shader.

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;

}

// Release the vertex shader.

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;

}

return;

}

void SkyDomeShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

// Get a pointer to the error message text buffer.

compileErrors = (char\*)(errorMessage->GetBufferPointer());

// Get the length of the message.

bufferSize = errorMessage->GetBufferSize();

// Open a file to write the error message to.

fout.open("shader-error.txt");

// Write out the error message.

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

// Close the file.

fout.close();

// Release the error message.

errorMessage->Release();

errorMessage = 0;

// Pop a message up on the screen to notify the user to check the text file for compile errors.

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;

}

bool SkyDomeShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, D3DXVECTOR4 apexColor, D3DXVECTOR4 centerColor)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

MatrixBufferType\* dataPtr;

GradientBufferType\* dataPtr2;

unsigned int bufferNumber;

// Transpose the matrices to prepare them for the shader.

D3DXMatrixTranspose(&worldMatrix, &worldMatrix);

D3DXMatrixTranspose(&viewMatrix, &viewMatrix);

D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

// Lock the constant buffer so it can be written to.

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr = (MatrixBufferType\*)mappedResource.pData;

// Copy the matrices into the constant buffer.

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

// Unlock the constant buffer.

deviceContext->Unmap(m\_matrixBuffer, 0);

// Set the position of the constant buffer in the vertex shader.

bufferNumber = 0;

// Finally set the constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

This is were we set the apex and center color variables in the pixel shader. They are set in the gradient constant buffer and then will be accessible in the pixel shader for rendering.

// Lock the gradient constant buffer so it can be written to.

result = deviceContext->Map(m\_gradientBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr2 = (GradientBufferType\*)mappedResource.pData;

// Copy the gradient color variables into the constant buffer.

dataPtr2->apexColor = apexColor;

dataPtr2->centerColor = centerColor;

// Unlock the constant buffer.

deviceContext->Unmap(m\_gradientBuffer, 0);

// Set the position of the gradient constant buffer in the pixel shader.

bufferNumber = 0;

// Finally set the gradient constant buffer in the pixel shader with the updated values.

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_gradientBuffer);

return true;

}

void SkyDomeShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

// Set the vertex input layout.

deviceContext->IASetInputLayout(m\_layout);

// Set the vertex and pixel shaders that will be used to render the triangles.

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

// Render the triangle.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}

D3dclass.h

The D3DClass has been updated to allow turning back face culling on and off.

////////////////////////////////////////////////////////////////////////////////

// Filename: d3dclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_D3DCLASS\_H\_

#define \_D3DCLASS\_H\_

/////////////

// LINKING //

/////////////

#pragma comment(lib, "dxgi.lib")

#pragma comment(lib, "d3d11.lib")

#pragma comment(lib, "d3dx11.lib")

#pragma comment(lib, "d3dx10.lib")

//////////////

// INCLUDES //

//////////////

#include <dxgi.h>

#include <d3dcommon.h>

#include <d3d11.h>

#include <d3dx10math.h>

////////////////////////////////////////////////////////////////////////////////

// Class name: D3DClass

////////////////////////////////////////////////////////////////////////////////

class D3DClass

{

public:

D3DClass();

D3DClass(const D3DClass&);

~D3DClass();

bool Initialize(int, int, bool, HWND, bool, float, float);

void Shutdown();

void BeginScene(float, float, float, float);

void EndScene();

ID3D11Device\* GetDevice();

ID3D11DeviceContext\* GetDeviceContext();

void GetProjectionMatrix(D3DXMATRIX&);

void GetWorldMatrix(D3DXMATRIX&);

void GetOrthoMatrix(D3DXMATRIX&);

void GetVideoCardInfo(char\*, int&);

void TurnZBufferOn();

void TurnZBufferOff();

void TurnOnAlphaBlending();

void TurnOffAlphaBlending();

These two functions are used for turning on and off back face culling.

void TurnOnCulling();

void TurnOffCulling();

private:

bool m\_vsync\_enabled;

int m\_videoCardMemory;

char m\_videoCardDescription[128];

IDXGISwapChain\* m\_swapChain;

ID3D11Device\* m\_device;

ID3D11DeviceContext\* m\_deviceContext;

ID3D11RenderTargetView\* m\_renderTargetView;

ID3D11Texture2D\* m\_depthStencilBuffer;

ID3D11DepthStencilState\* m\_depthStencilState;

ID3D11DepthStencilView\* m\_depthStencilView;

ID3D11RasterizerState\* m\_rasterState;

We also have a new rasterizer state for turning off back face culling.

ID3D11RasterizerState\* m\_rasterStateNoCulling;

D3DXMATRIX m\_projectionMatrix;

D3DXMATRIX m\_worldMatrix;

D3DXMATRIX m\_orthoMatrix;

ID3D11DepthStencilState\* m\_depthDisabledStencilState;

ID3D11BlendState\* m\_alphaEnableBlendingState;

ID3D11BlendState\* m\_alphaDisableBlendingState;

};

#endif

D3dclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: d3dclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "d3dclass.h"

D3DClass::D3DClass()

{

m\_swapChain = 0;

m\_device = 0;

m\_deviceContext = 0;

m\_renderTargetView = 0;

m\_depthStencilBuffer = 0;

m\_depthStencilState = 0;

m\_depthStencilView = 0;

m\_rasterState = 0;

m\_rasterStateNoCulling = 0;

m\_depthDisabledStencilState = 0;

m\_alphaEnableBlendingState = 0;

m\_alphaDisableBlendingState = 0;

}

D3DClass::D3DClass(const D3DClass& other)

{

}

D3DClass::~D3DClass()

{

}

bool D3DClass::Initialize(int screenWidth, int screenHeight, bool vsync, HWND hwnd, bool fullscreen, float screenDepth, float screenNear)

{

HRESULT result;

IDXGIFactory\* factory;

IDXGIAdapter\* adapter;

IDXGIOutput\* adapterOutput;

unsigned int numModes, i, numerator, denominator, stringLength;

DXGI\_MODE\_DESC\* displayModeList;

DXGI\_ADAPTER\_DESC adapterDesc;

int error;

DXGI\_SWAP\_CHAIN\_DESC swapChainDesc;

D3D\_FEATURE\_LEVEL featureLevel;

ID3D11Texture2D\* backBufferPtr;

D3D11\_TEXTURE2D\_DESC depthBufferDesc;

D3D11\_DEPTH\_STENCIL\_DESC depthStencilDesc;

D3D11\_DEPTH\_STENCIL\_VIEW\_DESC depthStencilViewDesc;

D3D11\_RASTERIZER\_DESC rasterDesc;

D3D11\_VIEWPORT viewport;

float fieldOfView, screenAspect;

D3D11\_DEPTH\_STENCIL\_DESC depthDisabledStencilDesc;

D3D11\_BLEND\_DESC blendStateDescription;

// Store the vsync setting.

m\_vsync\_enabled = vsync;

// Create a DirectX graphics interface factory.

result = CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory);

if(FAILED(result))

{

return false;

}

// Use the factory to create an adapter for the primary graphics interface (video card).

result = factory->EnumAdapters(0, &adapter);

if(FAILED(result))

{

return false;

}

// Enumerate the primary adapter output (monitor).

result = adapter->EnumOutputs(0, &adapterOutput);

if(FAILED(result))

{

return false;

}

// Get the number of modes that fit the DXGI\_FORMAT\_R8G8B8A8\_UNORM display format for the adapter output (monitor).

result = adapterOutput->GetDisplayModeList(DXGI\_FORMAT\_R8G8B8A8\_UNORM, DXGI\_ENUM\_MODES\_INTERLACED, &numModes, NULL);

if(FAILED(result))

{

return false;

}

// Create a list to hold all the possible display modes for this monitor/video card combination.

displayModeList = new DXGI\_MODE\_DESC[numModes];

if(!displayModeList)

{

return false;

}

// Now fill the display mode list structures.

result = adapterOutput->GetDisplayModeList(DXGI\_FORMAT\_R8G8B8A8\_UNORM, DXGI\_ENUM\_MODES\_INTERLACED, &numModes, displayModeList);

if(FAILED(result))

{

return false;

}

// Now go through all the display modes and find the one that matches the screen width and height.

// When a match is found store the numerator and denominator of the refresh rate for that monitor.

for(i=0; i<numModes; i++)

{

if(displayModeList[i].Width == (unsigned int)screenWidth)

{

if(displayModeList[i].Height == (unsigned int)screenHeight)

{

numerator = displayModeList[i].RefreshRate.Numerator;

denominator = displayModeList[i].RefreshRate.Denominator;

}

}

}

// Get the adapter (video card) description.

result = adapter->GetDesc(&adapterDesc);

if(FAILED(result))

{

return false;

}

// Store the dedicated video card memory in megabytes.

m\_videoCardMemory = (int)(adapterDesc.DedicatedVideoMemory / 1024 / 1024);

// Convert the name of the video card to a character array and store it.

error = wcstombs\_s(&stringLength, m\_videoCardDescription, 128, adapterDesc.Description, 128);

if(error != 0)

{

return false;

}

// Release the display mode list.

delete [] displayModeList;

displayModeList = 0;

// Release the adapter output.

adapterOutput->Release();

adapterOutput = 0;

// Release the adapter.

adapter->Release();

adapter = 0;

// Release the factory.

factory->Release();

factory = 0;

// Initialize the swap chain description.

ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));

// Set to a single back buffer.

swapChainDesc.BufferCount = 1;

// Set the width and height of the back buffer.

swapChainDesc.BufferDesc.Width = screenWidth;

swapChainDesc.BufferDesc.Height = screenHeight;

// Set regular 32-bit surface for the back buffer.

swapChainDesc.BufferDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

// Set the refresh rate of the back buffer.

if(m\_vsync\_enabled)

{

swapChainDesc.BufferDesc.RefreshRate.Numerator = numerator;

swapChainDesc.BufferDesc.RefreshRate.Denominator = denominator;

}

else

{

swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;

swapChainDesc.BufferDesc.RefreshRate.Denominator = 1;

}

// Set the usage of the back buffer.

swapChainDesc.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

// Set the handle for the window to render to.

swapChainDesc.OutputWindow = hwnd;

// Turn multisampling off.

swapChainDesc.SampleDesc.Count = 1;

swapChainDesc.SampleDesc.Quality = 0;

// Set to full screen or windowed mode.

if(fullscreen)

{

swapChainDesc.Windowed = false;

}

else

{

swapChainDesc.Windowed = true;

}

// Set the scan line ordering and scaling to unspecified.

swapChainDesc.BufferDesc.ScanlineOrdering = DXGI\_MODE\_SCANLINE\_ORDER\_UNSPECIFIED;

swapChainDesc.BufferDesc.Scaling = DXGI\_MODE\_SCALING\_UNSPECIFIED;

// Discard the back buffer contents after presenting.

swapChainDesc.SwapEffect = DXGI\_SWAP\_EFFECT\_DISCARD;

// Don't set the advanced flags.

swapChainDesc.Flags = 0;

// Set the feature level to DirectX 11.

featureLevel = D3D\_FEATURE\_LEVEL\_11\_0;

// Create the swap chain, Direct3D device, and Direct3D device context.

result = D3D11CreateDeviceAndSwapChain(NULL, D3D\_DRIVER\_TYPE\_HARDWARE, NULL, 0, &featureLevel, 1,

D3D11\_SDK\_VERSION, &swapChainDesc, &m\_swapChain, &m\_device, NULL, &m\_deviceContext);

if(FAILED(result))

{

return false;

}

// Get the pointer to the back buffer.

result = m\_swapChain->GetBuffer(0, \_\_uuidof(ID3D11Texture2D), (LPVOID\*)&backBufferPtr);

if(FAILED(result))

{

return false;

}

// Create the render target view with the back buffer pointer.

result = m\_device->CreateRenderTargetView(backBufferPtr, NULL, &m\_renderTargetView);

if(FAILED(result))

{

return false;

}

// Release pointer to the back buffer as we no longer need it.

backBufferPtr->Release();

backBufferPtr = 0;

// Initialize the description of the depth buffer.

ZeroMemory(&depthBufferDesc, sizeof(depthBufferDesc));

// Set up the description of the depth buffer.

depthBufferDesc.Width = screenWidth;

depthBufferDesc.Height = screenHeight;

depthBufferDesc.MipLevels = 1;

depthBufferDesc.ArraySize = 1;

depthBufferDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

depthBufferDesc.SampleDesc.Count = 1;

depthBufferDesc.SampleDesc.Quality = 0;

depthBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

depthBufferDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

depthBufferDesc.CPUAccessFlags = 0;

depthBufferDesc.MiscFlags = 0;

// Create the texture for the depth buffer using the filled out description.

result = m\_device->CreateTexture2D(&depthBufferDesc, NULL, &m\_depthStencilBuffer);

if(FAILED(result))

{

return false;

}

// Initialize the description of the stencil state.

ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));

// Set up the description of the stencil state.

depthStencilDesc.DepthEnable = true;

depthStencilDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;

depthStencilDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

depthStencilDesc.StencilEnable = true;

depthStencilDesc.StencilReadMask = 0xFF;

depthStencilDesc.StencilWriteMask = 0xFF;

// Stencil operations if pixel is front-facing.

depthStencilDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INCR;

depthStencilDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

// Stencil operations if pixel is back-facing.

depthStencilDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_DECR;

depthStencilDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

// Create the depth stencil state.

result = m\_device->CreateDepthStencilState(&depthStencilDesc, &m\_depthStencilState);

if(FAILED(result))

{

return false;

}

// Set the depth stencil state.

m\_deviceContext->OMSetDepthStencilState(m\_depthStencilState, 1);

// Initialize the depth stencil view.

ZeroMemory(&depthStencilViewDesc, sizeof(depthStencilViewDesc));

// Set up the depth stencil view description.

depthStencilViewDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

depthStencilViewDesc.ViewDimension = D3D11\_DSV\_DIMENSION\_TEXTURE2D;

depthStencilViewDesc.Texture2D.MipSlice = 0;

// Create the depth stencil view.

result = m\_device->CreateDepthStencilView(m\_depthStencilBuffer, &depthStencilViewDesc, &m\_depthStencilView);

if(FAILED(result))

{

return false;

}

// Bind the render target view and depth stencil buffer to the output render pipeline.

m\_deviceContext->OMSetRenderTargets(1, &m\_renderTargetView, m\_depthStencilView);

// Setup the raster description which will determine how and what polygons will be drawn.

rasterDesc.AntialiasedLineEnable = false;

rasterDesc.CullMode = D3D11\_CULL\_BACK;

rasterDesc.DepthBias = 0;

rasterDesc.DepthBiasClamp = 0.0f;

rasterDesc.DepthClipEnable = true;

rasterDesc.FillMode = D3D11\_FILL\_SOLID;

rasterDesc.FrontCounterClockwise = false;

rasterDesc.MultisampleEnable = false;

rasterDesc.ScissorEnable = false;

rasterDesc.SlopeScaledDepthBias = 0.0f;

// Create the rasterizer state from the description we just filled out.

result = m\_device->CreateRasterizerState(&rasterDesc, &m\_rasterState);

if(FAILED(result))

{

return false;

}

// Now set the rasterizer state.

m\_deviceContext->RSSetState(m\_rasterState);

Here is where we setup the rasterizer state for no back face culling.

// Setup a raster description which turns off back face culling.

rasterDesc.AntialiasedLineEnable = false;

rasterDesc.CullMode = D3D11\_CULL\_NONE;

rasterDesc.DepthBias = 0;

rasterDesc.DepthBiasClamp = 0.0f;

rasterDesc.DepthClipEnable = true;

rasterDesc.FillMode = D3D11\_FILL\_SOLID;

rasterDesc.FrontCounterClockwise = false;

rasterDesc.MultisampleEnable = false;

rasterDesc.ScissorEnable = false;

rasterDesc.SlopeScaledDepthBias = 0.0f;

// Create the no culling rasterizer state.

result = m\_device->CreateRasterizerState(&rasterDesc, &m\_rasterStateNoCulling);

if(FAILED(result))

{

return false;

}

// Setup the viewport for rendering.

viewport.Width = (float)screenWidth;

viewport.Height = (float)screenHeight;

viewport.MinDepth = 0.0f;

viewport.MaxDepth = 1.0f;

viewport.TopLeftX = 0.0f;

viewport.TopLeftY = 0.0f;

// Create the viewport.

m\_deviceContext->RSSetViewports(1, &viewport);

// Setup the projection matrix.

fieldOfView = (float)D3DX\_PI / 4.0f;

screenAspect = (float)screenWidth / (float)screenHeight;

// Create the projection matrix for 3D rendering.

D3DXMatrixPerspectiveFovLH(&m\_projectionMatrix, fieldOfView, screenAspect, screenNear, screenDepth);

// Initialize the world matrix to the identity matrix.

D3DXMatrixIdentity(&m\_worldMatrix);

// Create an orthographic projection matrix for 2D rendering.

D3DXMatrixOrthoLH(&m\_orthoMatrix, (float)screenWidth, (float)screenHeight, screenNear, screenDepth);

// Clear the second depth stencil state before setting the parameters.

ZeroMemory(&depthDisabledStencilDesc, sizeof(depthDisabledStencilDesc));

// Now create a second depth stencil state which turns off the Z buffer for 2D rendering. The only difference is

// that DepthEnable is set to false, all other parameters are the same as the other depth stencil state.

depthDisabledStencilDesc.DepthEnable = false;

depthDisabledStencilDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;

depthDisabledStencilDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

depthDisabledStencilDesc.StencilEnable = true;

depthDisabledStencilDesc.StencilReadMask = 0xFF;

depthDisabledStencilDesc.StencilWriteMask = 0xFF;

depthDisabledStencilDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INCR;

depthDisabledStencilDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

depthDisabledStencilDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_DECR;

depthDisabledStencilDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

// Create the state using the device.

result = m\_device->CreateDepthStencilState(&depthDisabledStencilDesc, &m\_depthDisabledStencilState);

if(FAILED(result))

{

return false;

}

// Clear the blend state description.

ZeroMemory(&blendStateDescription, sizeof(D3D11\_BLEND\_DESC));

// Create an alpha enabled blend state description.

blendStateDescription.RenderTarget[0].BlendEnable = TRUE;

blendStateDescription.RenderTarget[0].SrcBlend = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].DestBlend = D3D11\_BLEND\_INV\_SRC\_ALPHA;

blendStateDescription.RenderTarget[0].BlendOp = D3D11\_BLEND\_OP\_ADD;

blendStateDescription.RenderTarget[0].SrcBlendAlpha = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].DestBlendAlpha = D3D11\_BLEND\_ZERO;

blendStateDescription.RenderTarget[0].BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

blendStateDescription.RenderTarget[0].RenderTargetWriteMask = 0x0f;

// Create the blend state using the description.

result = m\_device->CreateBlendState(&blendStateDescription, &m\_alphaEnableBlendingState);

if(FAILED(result))

{

return false;

}

// Modify the description to create an alpha disabled blend state description.

blendStateDescription.RenderTarget[0].BlendEnable = FALSE;

// Create the second blend state using the description.

result = m\_device->CreateBlendState(&blendStateDescription, &m\_alphaDisableBlendingState);

if(FAILED(result))

{

return false;

}

return true;

}

void D3DClass::Shutdown()

{

// Before shutting down set to windowed mode or when you release the swap chain it will throw an exception.

if(m\_swapChain)

{

m\_swapChain->SetFullscreenState(false, NULL);

}

if(m\_alphaEnableBlendingState)

{

m\_alphaEnableBlendingState->Release();

m\_alphaEnableBlendingState = 0;

}

if(m\_alphaDisableBlendingState)

{

m\_alphaDisableBlendingState->Release();

m\_alphaDisableBlendingState = 0;

}

if(m\_depthDisabledStencilState)

{

m\_depthDisabledStencilState->Release();

m\_depthDisabledStencilState = 0;

}

if(m\_rasterStateNoCulling)

{

m\_rasterStateNoCulling->Release();

m\_rasterStateNoCulling = 0;

}

if(m\_rasterState)

{

m\_rasterState->Release();

m\_rasterState = 0;

}

if(m\_depthStencilView)

{

m\_depthStencilView->Release();

m\_depthStencilView = 0;

}

if(m\_depthStencilState)

{

m\_depthStencilState->Release();

m\_depthStencilState = 0;

}

if(m\_depthStencilBuffer)

{

m\_depthStencilBuffer->Release();

m\_depthStencilBuffer = 0;

}

if(m\_renderTargetView)

{

m\_renderTargetView->Release();

m\_renderTargetView = 0;

}

if(m\_deviceContext)

{

m\_deviceContext->Release();

m\_deviceContext = 0;

}

if(m\_device)

{

m\_device->Release();

m\_device = 0;

}

if(m\_swapChain)

{

m\_swapChain->Release();

m\_swapChain = 0;

}

return;

}

void D3DClass::BeginScene(float red, float green, float blue, float alpha)

{

float color[4];

// Setup the color to clear the buffer to.

color[0] = red;

color[1] = green;

color[2] = blue;

color[3] = alpha;

// Clear the back buffer.

m\_deviceContext->ClearRenderTargetView(m\_renderTargetView, color);

// Clear the depth buffer.

m\_deviceContext->ClearDepthStencilView(m\_depthStencilView, D3D11\_CLEAR\_DEPTH, 1.0f, 0);

return;

}

void D3DClass::EndScene()

{

// Present the back buffer to the screen since rendering is complete.

if(m\_vsync\_enabled)

{

// Lock to screen refresh rate.

m\_swapChain->Present(1, 0);

}

else

{

// Present as fast as possible.

m\_swapChain->Present(0, 0);

}

return;

}

ID3D11Device\* D3DClass::GetDevice()

{

return m\_device;

}

ID3D11DeviceContext\* D3DClass::GetDeviceContext()

{

return m\_deviceContext;

}

void D3DClass::GetProjectionMatrix(D3DXMATRIX& projectionMatrix)

{

projectionMatrix = m\_projectionMatrix;

return;

}

void D3DClass::GetWorldMatrix(D3DXMATRIX& worldMatrix)

{

worldMatrix = m\_worldMatrix;

return;

}

void D3DClass::GetOrthoMatrix(D3DXMATRIX& orthoMatrix)

{

orthoMatrix = m\_orthoMatrix;

return;

}

void D3DClass::GetVideoCardInfo(char\* cardName, int& memory)

{

strcpy\_s(cardName, 128, m\_videoCardDescription);

memory = m\_videoCardMemory;

return;

}

void D3DClass::TurnZBufferOn()

{

m\_deviceContext->OMSetDepthStencilState(m\_depthStencilState, 1);

return;

}

void D3DClass::TurnZBufferOff()

{

m\_deviceContext->OMSetDepthStencilState(m\_depthDisabledStencilState, 1);

return;

}

void D3DClass::TurnOnAlphaBlending()

{

float blendFactor[4];

// Setup the blend factor.

blendFactor[0] = 0.0f;

blendFactor[1] = 0.0f;

blendFactor[2] = 0.0f;

blendFactor[3] = 0.0f;

// Turn on the alpha blending.

m\_deviceContext->OMSetBlendState(m\_alphaEnableBlendingState, blendFactor, 0xffffffff);

return;

}

void D3DClass::TurnOffAlphaBlending()

{

float blendFactor[4];

// Setup the blend factor.

blendFactor[0] = 0.0f;

blendFactor[1] = 0.0f;

blendFactor[2] = 0.0f;

blendFactor[3] = 0.0f;

// Turn off the alpha blending.

m\_deviceContext->OMSetBlendState(m\_alphaDisableBlendingState, blendFactor, 0xffffffff);

return;

}

These are the two new helper functions that allow us to turn on and off back face culling.

void D3DClass::TurnOnCulling()

{

// Set the culling rasterizer state.

m\_deviceContext->RSSetState(m\_rasterState);

return;

}

void D3DClass::TurnOffCulling()

{

// Set the no back face culling rasterizer state.

m\_deviceContext->RSSetState(m\_rasterStateNoCulling);

return;

}

Applicationclass.h

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_APPLICATIONCLASS\_H\_

#define \_APPLICATIONCLASS\_H\_

/////////////

// GLOBALS //

/////////////

const bool FULL\_SCREEN = true;

const bool VSYNC\_ENABLED = true;

const float SCREEN\_DEPTH = 1000.0f;

const float SCREEN\_NEAR = 0.1f;

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "inputclass.h"

#include "d3dclass.h"

#include "cameraclass.h"

#include "terrainclass.h"

#include "timerclass.h"

#include "positionclass.h"

#include "fpsclass.h"

#include "cpuclass.h"

#include "fontshaderclass.h"

#include "textclass.h"

#include "terrainshaderclass.h"

#include "lightclass.h"

The two new header files for the SkyDomeClass and SkyDomeShaderClass are added here.

#include "skydomeclass.h"

#include "skydomeshaderclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: ApplicationClass

////////////////////////////////////////////////////////////////////////////////

class ApplicationClass

{

public:

ApplicationClass();

ApplicationClass(const ApplicationClass&);

~ApplicationClass();

bool Initialize(HINSTANCE, HWND, int, int);

void Shutdown();

bool Frame();

private:

bool HandleInput(float);

bool RenderGraphics();

private:

InputClass\* m\_Input;

D3DClass\* m\_Direct3D;

CameraClass\* m\_Camera;

TerrainClass\* m\_Terrain;

TimerClass\* m\_Timer;

PositionClass\* m\_Position;

FpsClass\* m\_Fps;

CpuClass\* m\_Cpu;

FontShaderClass\* m\_FontShader;

TextClass\* m\_Text;

TerrainShaderClass\* m\_TerrainShader;

LightClass\* m\_Light;

We add the SkyDomeClass and SkyDomeShaderClass object pointers here.

SkyDomeClass\* m\_SkyDome;

SkyDomeShaderClass\* m\_SkyDomeShader;

};

#endif

Applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "applicationclass.h"

ApplicationClass::ApplicationClass()

{

m\_Input = 0;

m\_Direct3D = 0;

m\_Camera = 0;

m\_Terrain = 0;

m\_Timer = 0;

m\_Position = 0;

m\_Fps = 0;

m\_Cpu = 0;

m\_FontShader = 0;

m\_Text = 0;

m\_TerrainShader = 0;

m\_Light = 0;

Initialize the new object pointers here in the class constructor.

m\_SkyDome = 0;

m\_SkyDomeShader = 0;

}

ApplicationClass::ApplicationClass(const ApplicationClass& other)

{

}

ApplicationClass::~ApplicationClass()

{

}

bool ApplicationClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)

{

bool result;

float cameraX, cameraY, cameraZ;

D3DXMATRIX baseViewMatrix;

char videoCard[128];

int videoMemory;

// Create the input object. The input object will be used to handle reading the keyboard and mouse input from the user.

m\_Input = new InputClass;

if(!m\_Input)

{

return false;

}

// Initialize the input object.

result = m\_Input->Initialize(hinstance, hwnd, screenWidth, screenHeight);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the input object.", L"Error", MB\_OK);

return false;

}

// Create the Direct3D object.

m\_Direct3D = new D3DClass;

if(!m\_Direct3D)

{

return false;

}

// Initialize the Direct3D object.

result = m\_Direct3D->Initialize(screenWidth, screenHeight, VSYNC\_ENABLED, hwnd, FULL\_SCREEN, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize DirectX 11.", L"Error", MB\_OK);

return false;

}

// Create the camera object.

m\_Camera = new CameraClass;

if(!m\_Camera)

{

return false;

}

// Initialize a base view matrix with the camera for 2D user interface rendering.

m\_Camera->SetPosition(0.0f, 0.0f, -1.0f);

m\_Camera->Render();

m\_Camera->GetViewMatrix(baseViewMatrix);

// Set the initial position of the camera.

cameraX = 50.0f;

cameraY = 2.0f;

cameraZ = -7.0f;

m\_Camera->SetPosition(cameraX, cameraY, cameraZ);

// Create the terrain object.

m\_Terrain = new TerrainClass;

if(!m\_Terrain)

{

return false;

}

// Initialize the terrain object.

result = m\_Terrain->Initialize(m\_Direct3D->GetDevice(), "../Engine/data/heightmap01.bmp", L"../Engine/data/dirt01.dds", "../Engine/data/colorm01.bmp");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain object.", L"Error", MB\_OK);

return false;

}

// Create the timer object.

m\_Timer = new TimerClass;

if(!m\_Timer)

{

return false;

}

// Initialize the timer object.

result = m\_Timer->Initialize();

if(!result)

{

MessageBox(hwnd, L"Could not initialize the timer object.", L"Error", MB\_OK);

return false;

}

// Create the position object.

m\_Position = new PositionClass;

if(!m\_Position)

{

return false;

}

// Set the initial position of the viewer to the same as the initial camera position.

m\_Position->SetPosition(cameraX, cameraY, cameraZ);

// Create the fps object.

m\_Fps = new FpsClass;

if(!m\_Fps)

{

return false;

}

// Initialize the fps object.

m\_Fps->Initialize();

// Create the cpu object.

m\_Cpu = new CpuClass;

if(!m\_Cpu)

{

return false;

}

// Initialize the cpu object.

m\_Cpu->Initialize();

// Create the font shader object.

m\_FontShader = new FontShaderClass;

if(!m\_FontShader)

{

return false;

}

// Initialize the font shader object.

result = m\_FontShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the font shader object.", L"Error", MB\_OK);

return false;

}

// Create the text object.

m\_Text = new TextClass;

if(!m\_Text)

{

return false;

}

// Initialize the text object.

result = m\_Text->Initialize(m\_Direct3D->GetDevice(), m\_Direct3D->GetDeviceContext(), hwnd, screenWidth, screenHeight, baseViewMatrix);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the text object.", L"Error", MB\_OK);

return false;

}

// Retrieve the video card information.

m\_Direct3D->GetVideoCardInfo(videoCard, videoMemory);

// Set the video card information in the text object.

result = m\_Text->SetVideoCardInfo(videoCard, videoMemory, m\_Direct3D->GetDeviceContext());

if(!result)

{

MessageBox(hwnd, L"Could not set video card info in the text object.", L"Error", MB\_OK);

return false;

}

// Create the terrain shader object.

m\_TerrainShader = new TerrainShaderClass;

if(!m\_TerrainShader)

{

return false;

}

// Initialize the terrain shader object.

result = m\_TerrainShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain shader object.", L"Error", MB\_OK);

return false;

}

// Create the light object.

m\_Light = new LightClass;

if(!m\_Light)

{

return false;

}

// Initialize the light object.

m\_Light->SetAmbientColor(0.05f, 0.05f, 0.05f, 1.0f);

m\_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);

m\_Light->SetDirection(-0.5f, -1.0f, 0.0f);

Create and initialize both the SkyDomeClass and SkyDomeShaderClass objects here.

// Create the sky dome object.

m\_SkyDome = new SkyDomeClass;

if(!m\_SkyDome)

{

return false;

}

// Initialize the sky dome object.

result = m\_SkyDome->Initialize(m\_Direct3D->GetDevice());

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome object.", L"Error", MB\_OK);

return false;

}

// Create the sky dome shader object.

m\_SkyDomeShader = new SkyDomeShaderClass;

if(!m\_SkyDomeShader)

{

return false;

}

// Initialize the sky dome shader object.

result = m\_SkyDomeShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome shader object.", L"Error", MB\_OK);

return false;

}

return true;

}

void ApplicationClass::Shutdown()

{

Release the SkyDomeClass and SkyDomeShaderClass objects here in the Shutdown function.

// Release the sky dome shader object.

if(m\_SkyDomeShader)

{

m\_SkyDomeShader->Shutdown();

delete m\_SkyDomeShader;

m\_SkyDomeShader = 0;

}

// Release the sky dome object.

if(m\_SkyDome)

{

m\_SkyDome->Shutdown();

delete m\_SkyDome;

m\_SkyDome = 0;

}

// Release the light object.

if(m\_Light)

{

delete m\_Light;

m\_Light = 0;

}

// Release the terrain shader object.

if(m\_TerrainShader)

{

m\_TerrainShader->Shutdown();

delete m\_TerrainShader;

m\_TerrainShader = 0;

}

// Release the text object.

if(m\_Text)

{

m\_Text->Shutdown();

delete m\_Text;

m\_Text = 0;

}

// Release the font shader object.

if(m\_FontShader)

{

m\_FontShader->Shutdown();

delete m\_FontShader;

m\_FontShader = 0;

}

// Release the cpu object.

if(m\_Cpu)

{

m\_Cpu->Shutdown();

delete m\_Cpu;

m\_Cpu = 0;

}

// Release the fps object.

if(m\_Fps)

{

delete m\_Fps;

m\_Fps = 0;

}

// Release the position object.

if(m\_Position)

{

delete m\_Position;

m\_Position = 0;

}

// Release the timer object.

if(m\_Timer)

{

delete m\_Timer;

m\_Timer = 0;

}

// Release the terrain object.

if(m\_Terrain)

{

m\_Terrain->Shutdown();

delete m\_Terrain;

m\_Terrain = 0;

}

// Release the camera object.

if(m\_Camera)

{

delete m\_Camera;

m\_Camera = 0;

}

// Release the Direct3D object.

if(m\_Direct3D)

{

m\_Direct3D->Shutdown();

delete m\_Direct3D;

m\_Direct3D = 0;

}

// Release the input object.

if(m\_Input)

{

m\_Input->Shutdown();

delete m\_Input;

m\_Input = 0;

}

return;

}

bool ApplicationClass::Frame()

{

bool result;

// Read the user input.

result = m\_Input->Frame();

if(!result)

{

return false;

}

// Check if the user pressed escape and wants to exit the application.

if(m\_Input->IsEscapePressed() == true)

{

return false;

}

// Update the system stats.

m\_Timer->Frame();

m\_Fps->Frame();

m\_Cpu->Frame();

// Update the FPS value in the text object.

result = m\_Text->SetFps(m\_Fps->GetFps(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the CPU usage value in the text object.

result = m\_Text->SetCpu(m\_Cpu->GetCpuPercentage(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Do the frame input processing.

result = HandleInput(m\_Timer->GetTime());

if(!result)

{

return false;

}

// Render the graphics.

result = RenderGraphics();

if(!result)

{

return false;

}

return result;

}

bool ApplicationClass::HandleInput(float frameTime)

{

bool keyDown, result;

float posX, posY, posZ, rotX, rotY, rotZ;

// Set the frame time for calculating the updated position.

m\_Position->SetFrameTime(frameTime);

// Handle the input.

keyDown = m\_Input->IsLeftPressed();

m\_Position->TurnLeft(keyDown);

keyDown = m\_Input->IsRightPressed();

m\_Position->TurnRight(keyDown);

keyDown = m\_Input->IsUpPressed();

m\_Position->MoveForward(keyDown);

keyDown = m\_Input->IsDownPressed();

m\_Position->MoveBackward(keyDown);

keyDown = m\_Input->IsAPressed();

m\_Position->MoveUpward(keyDown);

keyDown = m\_Input->IsZPressed();

m\_Position->MoveDownward(keyDown);

keyDown = m\_Input->IsPgUpPressed();

m\_Position->LookUpward(keyDown);

keyDown = m\_Input->IsPgDownPressed();

m\_Position->LookDownward(keyDown);

// Get the view point position/rotation.

m\_Position->GetPosition(posX, posY, posZ);

m\_Position->GetRotation(rotX, rotY, rotZ);

// Set the position of the camera.

m\_Camera->SetPosition(posX, posY, posZ);

m\_Camera->SetRotation(rotX, rotY, rotZ);

// Update the position values in the text object.

result = m\_Text->SetCameraPosition(posX, posY, posZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the rotation values in the text object.

result = m\_Text->SetCameraRotation(rotX, rotY, rotZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

return true;

}

bool ApplicationClass::RenderGraphics()

{

D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix;

D3DXVECTOR3 cameraPosition;

bool result;

// Clear the scene.

m\_Direct3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

// Generate the view matrix based on the camera's position.

m\_Camera->Render();

// Get the world, view, projection, and ortho matrices from the camera and Direct3D objects.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

m\_Direct3D->GetOrthoMatrix(orthoMatrix);

Here is where we get the position of the camera and then translate our sky dome to be centered around the camera position.

// Get the position of the camera.

cameraPosition = m\_Camera->GetPosition();

// Translate the sky dome to be centered around the camera position.

D3DXMatrixTranslation(&worldMatrix, cameraPosition.x, cameraPosition.y, cameraPosition.z);

Before rendering the sky dome we turn off both back face culling and the Z buffer.

// Turn off back face culling.

m\_Direct3D->TurnOffCulling();

// Turn off the Z buffer.

m\_Direct3D->TurnZBufferOff();

We then render the sky dome using the sky dome shader.

// Render the sky dome using the sky dome shader.

m\_SkyDome->Render(m\_Direct3D->GetDeviceContext());

m\_SkyDomeShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyDome->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyDome->GetApexColor(), m\_SkyDome->GetCenterColor());

Once rendering is complete we turn back face culling and the Z buffer on again and resume rendering the rest of the scene as normal.

// Turn back face culling back on.

m\_Direct3D->TurnOnCulling();

// Turn the Z buffer back on.

m\_Direct3D->TurnZBufferOn();

// Reset the world matrix.

m\_Direct3D->GetWorldMatrix(worldMatrix);

// Render the terrain buffers.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

// Render the terrain using the terrain shader.

result = m\_TerrainShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_Light->GetAmbientColor(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(), m\_Terrain->GetTexture());

if(!result)

{

return false;

}

// Turn off the Z buffer to begin all 2D rendering.

m\_Direct3D->TurnZBufferOff();

// Turn on the alpha blending before rendering the text.

m\_Direct3D->TurnOnAlphaBlending();

// Render the text user interface elements.

result = m\_Text->Render(m\_Direct3D->GetDeviceContext(), m\_FontShader, worldMatrix, orthoMatrix);

if(!result)

{

return false;

}

// Turn off alpha blending after rendering the text.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on now that all 2D rendering has completed.

m\_Direct3D->TurnZBufferOn();

// Present the rendered scene to the screen.

m\_Direct3D->EndScene();

return true;

}

Summary

We now have a colored sky background for the terrain using a very simple shader and a low polygon sphere object.

To Do Exercises

1. Compile the code and run the program. Use the page up and page down keys to look at the sky dome color gradient.

2. Change the apex and center color in the skydomeclass.cpp file inside the Initialize function to see the effect it has on the color gradient.

3. Set the fill mode in the no back face culling raster state to D3D11\_FILL\_WIREFRAME to see how the sky dome moves with the camera.

4. Create multiple gradients in the sky dome. For example have a gradient from +1.0 to +0.5, and then one from +0.25 to +0.0, and one from 0.0 and below. Do this by creating multiple "if" statements using the height variable and then supply several gradient colors to the pixel shader by expanding the GradientBuffer constant buffer.

5. Modify the pixel shader so the gradient goes from east to west.

6. Create a list of gradient colors for each hour in the day. Then create a system that keeps track of the time and updates your shader with the current gradient for that time of the day. Speed up the time so you can see the transitions. Note you may need to do a 30 second transition/interpolation at each hour point to prevent the popping effect created by the sudden gradient transition.

7. Increase and decrease the polygon count of the sphere used for the sky dome. Observe the effect it has on the look of the gradient and the dome.

Source Code

Source Code and Data Files: [tersrc10.zip](http://web.archive.org/web/20140722202020/http:/rastertek.com/tersrc10.zip)

Executable: [terexe10.zip](http://web.archive.org/web/20140722202020/http:/rastertek.com/terexe10.zip)

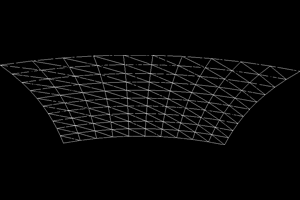
**Tutorial 11: Bitmap Clouds**

http://web.archive.org/web/20131111194909/http:/rastertek.com/pic1001.gif

This DirectX 11 terrain tutorial will cover how to implement clouds. The particular implementation will be bitmap clouds which is one of the simpler methods to use. The code in this tutorial builds off the previous sky dome tutorial as the sky dome is required for coloring the clouds.

To implement clouds we are going to first need some geometry to render the cloud bitmaps onto. The ideal geometry to use is a plane that is slightly curved. You can create one in a modeling program but since they are very easy to build we will implement the geometry in the code so that we can modify it as needed. The plane model will be encapsulated in a class called SkyPlaneClass.

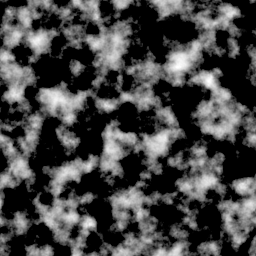
And just like the sky dome in the previous tutorial we will always center the sky plane at the camera's location and place it just slightly above us. This way as the camera moves in the world the sky plane moves with it. For rendering the sky plane we will turn off the depth buffer so that it overwrites and combines with everything behind it giving the illusion that the geometry is much larger than what it actually is. This was the same approach used with the sky dome. And since we are generating the plane in the code we won't need to turn off back face culling as we will create the plane facing downwards.



Note that the edges of the sky plane will be visible so you will need to adjust for this in your own code. Most people deal with the issue by ensuring there is a surrounding band of terrain that the camera will never go above so the edges are hidden. Other people use walls, buildings, and so forth. Another approach is to use a third alpha texture to create a fade effect at the edges so it looks like it blends into the horizon but it doesn't always look correct when you do that.

The clouds that we will be using are going to be two different bitmaps that we will render onto the sky plane. The reason that we are using two instead of one is so that we can translate them at different speeds creating the illusion of two separate cloud layers with one higher than the other. The speed you translate each at is key in creating the perception that one layer is much higher than the other, even though they are rendered onto the same geometry.





To color the clouds we simply turn on additive blending and make sure that the sky dome is rendered first. This way the clouds will blend with the sky dome's color.

We will begin the code section by examining the SkyPlaneClass first.

Skyplaneclass.h

The SkyPlaneClass encapsulates everything related to the plane used for rendering the clouds. It holds the geometry for the sky plane, the two bitmaps textures for the clouds, and all the variables for the shader that relate to how to draw the sky plane.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SKYPLANECLASS\_H\_

#define \_SKYPLANECLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "textureclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: SkyPlaneClass

////////////////////////////////////////////////////////////////////////////////

class SkyPlaneClass

{

private:

The SkyPlaneType structure is used for storing the sky plane geometry. We generate position and texture coordinates for the plane and then store them in an array of SkyPlaneType. There are no normals since the clouds use the sky dome for color and lighting appearance.

struct SkyPlaneType

{

float x, y, z;

float tu, tv;

};

The VertexType requires position and texture coordinates for rendering the sky plane.

struct VertexType

{

D3DXVECTOR3 position;

D3DXVECTOR2 texture;

};

public:

SkyPlaneClass();

SkyPlaneClass(const SkyPlaneClass&);

~SkyPlaneClass();

bool Initialize(ID3D11Device\*, WCHAR\*, WCHAR\*);

void Shutdown();

void Render(ID3D11DeviceContext\*);

void Frame();

int GetIndexCount();

ID3D11ShaderResourceView\* GetCloudTexture1();

ID3D11ShaderResourceView\* GetCloudTexture2();

float GetBrightness();

float GetTranslation(int);

private:

bool InitializeSkyPlane(int, float, float, float, int);

void ShutdownSkyPlane();

bool InitializeBuffers(ID3D11Device\*, int);

void ShutdownBuffers();

void RenderBuffers(ID3D11DeviceContext\*);

bool LoadTextures(ID3D11Device\*, WCHAR\*, WCHAR\*);

void ReleaseTextures();

private:

The m\_skyPlane array is used to hold the plane geometry.

SkyPlaneType\* m\_skyPlane;

int m\_vertexCount, m\_indexCount;

ID3D11Buffer \*m\_vertexBuffer, \*m\_indexBuffer;

The sky plane will use two cloud textures that are rendered to it.

TextureClass \*m\_CloudTexture1, \*m\_CloudTexture2;

The brightness of the clouds is stored here and set in the pixel shader during rendering.

float m\_brightness;

The cloud location and speed are stored in these two arrays.

float m\_translationSpeed[4];

float m\_textureTranslation[4];

};

#endif

Skyplaneclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "skyplaneclass.h"

We set the class private member pointers to null in the class constructor.

SkyPlaneClass::SkyPlaneClass()

{

m\_skyPlane = 0;

m\_vertexBuffer = 0;

m\_indexBuffer = 0;

m\_CloudTexture1 = 0;

m\_CloudTexture2 = 0;

}

SkyPlaneClass::SkyPlaneClass(const SkyPlaneClass& other)

{

}

SkyPlaneClass::~SkyPlaneClass()

{

}

The Initialize function is where we do all the setup for the sky plane. It takes as input the two cloud texture file names as well as the Direct3D device.

bool SkyPlaneClass::Initialize(ID3D11Device\* device, WCHAR\* textureFilename1, WCHAR\* textureFilename2)

{

int skyPlaneResolution, textureRepeat;

float skyPlaneWidth, skyPlaneTop, skyPlaneBottom;

bool result;

Here is where we set the sky plane parameters that are used for generating the plane geometry. The skyPlaneResolution is used for specifying how many quads that sky plane should be composed of in the X and Z direction, increasing this value makes it higher poly and smoother. The skyPlaneWidth is the length of the plane.

The skyPlaneTop and skyPlaneBottom represent the height and base of the curved plane. The bottom four corners of the plane will be at skyPlaneBottom and the center of the plane will be at skyPlaneTop. All other points are interpolated between those two values.

The textureRepeat value determines how many times to repeat the texture over the sky plane. This is used to generate the UV coordinates. Note that a single 256x256 texture mapped just once over the sky plane will look fairly pixelated, therefore you either want to increase the texture size or map it multiple times to reduce how pixelated it looks. Note that the higher the resolution this program runs at the higher resolution the texture will need to be to look non-pixelated (likewise how many times it needs to be repeated).

// Set the sky plane parameters.

skyPlaneResolution = 10;

skyPlaneWidth = 10.0f;

skyPlaneTop = 0.5f;

skyPlaneBottom = 0.0f;

textureRepeat = 4;

Setting the brightness is important for making clouds look realistic when using bitmaps that range from just 0 to 255. The brightness value lowers how white the clouds are which allows you to give them more of a faded look just like real clouds have. The value range here is from 0.0f to 1.0f. I have set it to 0.65f so that the clouds are at 65% brightness.

// Set the brightness of the clouds.

m\_brightness = 0.65f;

The translation speed is how fast we translate the cloud textures over the sky plane. Each cloud can be translated on both the X and Z axis. There are two textures so we store the dual speed for both in a 4 float array.

// Setup the cloud translation speed increments.

m\_translationSpeed[0] = 0.0003f; // First texture X translation speed.

m\_translationSpeed[1] = 0.0f; // First texture Z translation speed.

m\_translationSpeed[2] = 0.00015f; // Second texture X translation speed.

m\_translationSpeed[3] = 0.0f; // Second texture Z translation speed.

We also store the current translation for the two textures and provide it to the pixel shader during rendering.

// Initialize the texture translation values.

m\_textureTranslation[0] = 0.0f;

m\_textureTranslation[1] = 0.0f;

m\_textureTranslation[2] = 0.0f;

m\_textureTranslation[3] = 0.0f;

Once all our values are set we then create the sky plane, load it into a vertex and index buffer, and then load the textures. Note that I would generally make all the sky plane parameters as input to the Initialize function but it is easier for the tutorial explanation and for modifying them at first by having them inside the function for now.

// Create the sky plane.

result = InitializeSkyPlane(skyPlaneResolution, skyPlaneWidth, skyPlaneTop, skyPlaneBottom, textureRepeat);

if(!result)

{

return false;

}

// Create the vertex and index buffer for the sky plane.

result = InitializeBuffers(device, skyPlaneResolution);

if(!result)

{

return false;

}

// Load the sky plane textures.

result = LoadTextures(device, textureFilename1, textureFilename2);

if(!result)

{

return false;

}

return true;

}

The Shutdown function is where we release the sky plane, the buffers, and the textures.

void SkyPlaneClass::Shutdown()

{

// Release the sky plane textures.

ReleaseTextures();

// Release the vertex and index buffer that were used for rendering the sky plane.

ShutdownBuffers();

// Release the sky plane array.

ShutdownSkyPlane();

return;

}

The Render function calls RenderBuffers to put the sky plane geometry on the graphics pipeline for drawing.

void SkyPlaneClass::Render(ID3D11DeviceContext\* deviceContext)

{

// Render the sky plane.

RenderBuffers(deviceContext);

return;

}

The frame processing that we do for the sky plane is the cloud texture translation which simulates movement of the clouds across the sky. The coordinates are translated according to the speed given for that direction. Index 0 and 1 is for the X and Z on the first cloud. Index 2 and 3 is for the X and Z on the second cloud. We also truncate the values so they never go over 1.0f. Note that if you unlock the vsync the clouds will go at a speed according to the new frame rate, to avoid that you should pass in the frame time and adjust the translation accordingly.

void SkyPlaneClass::Frame()

{

// Increment the translation values to simulate the moving clouds.

m\_textureTranslation[0] += m\_translationSpeed[0];

m\_textureTranslation[1] += m\_translationSpeed[1];

m\_textureTranslation[2] += m\_translationSpeed[2];

m\_textureTranslation[3] += m\_translationSpeed[3];

// Keep the values in the zero to one range.

if(m\_textureTranslation[0] > 1.0f) { m\_textureTranslation[0] -= 1.0f; }

if(m\_textureTranslation[1] > 1.0f) { m\_textureTranslation[1] -= 1.0f; }

if(m\_textureTranslation[2] > 1.0f) { m\_textureTranslation[2] -= 1.0f; }

if(m\_textureTranslation[3] > 1.0f) { m\_textureTranslation[3] -= 1.0f; }

return;

}

GetIndexCount returns the number of indexes in the index buffer.

int SkyPlaneClass::GetIndexCount()

{

return m\_indexCount;

}

GetCloudTexture1 returns the first cloud texture resource.

ID3D11ShaderResourceView\* SkyPlaneClass::GetCloudTexture1()

{

return m\_CloudTexture1->GetTexture();

}

GetCloudTexture2 returns the second cloud texture resource.

ID3D11ShaderResourceView\* SkyPlaneClass::GetCloudTexture2()

{

return m\_CloudTexture2->GetTexture();

}

The GetBrightness function returns the current brightness value that we want applied to the clouds in the pixel shader.

float SkyPlaneClass::GetBrightness()

{

return m\_brightness;

}

The GetTranslation function returns the texture translation value for the given index.

float SkyPlaneClass::GetTranslation(int index)

{

return m\_textureTranslation[index];

}

InitializeSkyPlane is where we build the geometry for the sky plane. We first create an array to hold the geometry and then we setup the increment values needed to build the sky plane in the for loop. Then we run the for loop and create the position and texture coordinates for each vertex based on the increment values. This process builds the curved plane that we will use to render the clouds onto.

bool SkyPlaneClass::InitializeSkyPlane(int skyPlaneResolution, float skyPlaneWidth, float skyPlaneTop, float skyPlaneBottom, int textureRepeat)

{

float quadSize, radius, constant, textureDelta;

int i, j, index;

float positionX, positionY, positionZ, tu, tv;

// Create the array to hold the sky plane coordinates.

m\_skyPlane = new SkyPlaneType[(skyPlaneResolution + 1) \* (skyPlaneResolution + 1)];

if(!m\_skyPlane)

{

return false;

}

// Determine the size of each quad on the sky plane.

quadSize = skyPlaneWidth / (float)skyPlaneResolution;

// Calculate the radius of the sky plane based on the width.

radius = skyPlaneWidth / 2.0f;

// Calculate the height constant to increment by.

constant = (skyPlaneTop - skyPlaneBottom) / (radius \* radius);

// Calculate the texture coordinate increment value.

textureDelta = (float)textureRepeat / (float)skyPlaneResolution;

// Loop through the sky plane and build the coordinates based on the increment values given.

for(j=0; j<=skyPlaneResolution; j++)

{

for(i=0; i<=skyPlaneResolution; i++)

{

// Calculate the vertex coordinates.

positionX = (-0.5f \* skyPlaneWidth) + ((float)i \* quadSize);

positionZ = (-0.5f \* skyPlaneWidth) + ((float)j \* quadSize);

positionY = skyPlaneTop - (constant \* ((positionX \* positionX) + (positionZ \* positionZ)));

// Calculate the texture coordinates.

tu = (float)i \* textureDelta;

tv = (float)j \* textureDelta;

// Calculate the index into the sky plane array to add this coordinate.

index = j \* (skyPlaneResolution + 1) + i;

// Add the coordinates to the sky plane array.

m\_skyPlane[index].x = positionX;

m\_skyPlane[index].y = positionY;

m\_skyPlane[index].z = positionZ;

m\_skyPlane[index].tu = tu;

m\_skyPlane[index].tv = tv;

}

}

return true;

}

The ShutdownSkyPlane function releases the array that was holding the sky plane geometry.

void SkyPlaneClass::ShutdownSkyPlane()

{

// Release the sky plane array.

if(m\_skyPlane)

{

delete [] m\_skyPlane;

m\_skyPlane = 0;

}

return;

}

InitializeBuffers loads the array that was holding the sky plane geometry into a vertex and index buffer so that it can be rendered.

bool SkyPlaneClass::InitializeBuffers(ID3D11Device\* device, int skyPlaneResolution)

{

VertexType\* vertices;

unsigned long\* indices;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

int i, j, index, index1, index2, index3, index4;

// Calculate the number of vertices in the sky plane mesh.

m\_vertexCount = (skyPlaneResolution + 1) \* (skyPlaneResolution + 1) \* 6;

// Set the index count to the same as the vertex count.

m\_indexCount = m\_vertexCount;

// Create the vertex array.

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;

}

// Create the index array.

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;

}

// Initialize the index into the vertex array.

index = 0;

// Load the vertex and index array with the sky plane array data.

for(j=0; j<skyPlaneResolution; j++)

{

for(i=0; i<skyPlaneResolution; i++)

{

index1 = j \* (skyPlaneResolution + 1) + i;

index2 = j \* (skyPlaneResolution + 1) + (i+1);

index3 = (j+1) \* (skyPlaneResolution + 1) + i;

index4 = (j+1) \* (skyPlaneResolution + 1) + (i+1);

// Triangle 1 - Upper Left

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index1].x, m\_skyPlane[index1].y, m\_skyPlane[index1].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index1].tu, m\_skyPlane[index1].tv);

indices[index] = index;

index++;

// Triangle 1 - Upper Right

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index2].x, m\_skyPlane[index2].y, m\_skyPlane[index2].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index2].tu, m\_skyPlane[index2].tv);

indices[index] = index;

index++;

// Triangle 1 - Bottom Left

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index3].x, m\_skyPlane[index3].y, m\_skyPlane[index3].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index3].tu, m\_skyPlane[index3].tv);

indices[index] = index;

index++;

// Triangle 2 - Bottom Left

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index3].x, m\_skyPlane[index3].y, m\_skyPlane[index3].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index3].tu, m\_skyPlane[index3].tv);

indices[index] = index;

index++;

// Triangle 2 - Upper Right

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index2].x, m\_skyPlane[index2].y, m\_skyPlane[index2].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index2].tu, m\_skyPlane[index2].tv);

indices[index] = index;

index++;

// Triangle 2 - Bottom Right

vertices[index].position = D3DXVECTOR3(m\_skyPlane[index4].x, m\_skyPlane[index4].y, m\_skyPlane[index4].z);

vertices[index].texture = D3DXVECTOR2(m\_skyPlane[index4].tu, m\_skyPlane[index4].tv);

indices[index] = index;

index++;

}

}

// Set up the description of the vertex buffer.

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the vertex data.

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

// Now finally create the vertex buffer.

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{

return false;

}

// Set up the description of the index buffer.

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the index data.

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

// Create the index buffer.

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{

return false;

}

// Release the arrays now that the vertex and index buffers have been created and loaded.

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;

}

The ShutdownBuffers function releases the buffers that were used to render the sky plane.

void SkyPlaneClass::ShutdownBuffers()

{

// Release the index buffer.

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;

}

// Release the vertex buffer.

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;

}

return;

}

RenderBuffers puts the sky plane geometry on the graphics pipeline for rendering by the shader.

void SkyPlaneClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

// Set vertex buffer stride and offset.

stride = sizeof(VertexType);

offset = 0;

// Set the vertex buffer to active in the input assembler so it can be rendered.

deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

// Set the index buffer to active in the input assembler so it can be rendered.

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}

The LoadTextures function loads the two cloud textures that will be used for rendering with.

bool SkyPlaneClass::LoadTextures(ID3D11Device\* device, WCHAR\* textureFilename1, WCHAR\* textureFilename2)

{

bool result;

// Create the first cloud texture object.

m\_CloudTexture1 = new TextureClass;

if(!m\_CloudTexture1)

{

return false;

}

// Initialize the first cloud texture object.

result = m\_CloudTexture1->Initialize(device, textureFilename1);

if(!result)

{

return false;

}

// Create the second cloud texture object.

m\_CloudTexture2 = new TextureClass;

if(!m\_CloudTexture2)

{

return false;

}

// Initialize the second cloud texture object.

result = m\_CloudTexture2->Initialize(device, textureFilename2);

if(!result)

{

return false;

}

return true;

}

ReleaseTextures releases the two cloud textures that were used for rendering.

void SkyPlaneClass::ReleaseTextures()

{

// Release the texture objects.

if(m\_CloudTexture1)

{

m\_CloudTexture1->Shutdown();

delete m\_CloudTexture1;

m\_CloudTexture1 = 0;

}

if(m\_CloudTexture2)

{

m\_CloudTexture2->Shutdown();

delete m\_CloudTexture2;

m\_CloudTexture2 = 0;

}

return;

}

Skyplane.vs

The sky plane vertex shader is fairly generic and just sends through the transformed position and texture coordinates to the pixel shader.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplane.vs

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer MatrixBuffer

{

matrix worldMatrix;

matrix viewMatrix;

matrix projectionMatrix;

};

//////////////

// TYPEDEFS //

//////////////

struct VertexInputType

{

float4 position : POSITION;

float2 tex : TEXCOORD0;

};

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

};

////////////////////////////////////////////////////////////////////////////////

// Vertex Shader

////////////////////////////////////////////////////////////////////////////////

PixelInputType SkyPlaneVertexShader(VertexInputType input)

{

PixelInputType output;

// Change the position vector to be 4 units for proper matrix calculations.

input.position.w = 1.0f;

// Calculate the position of the vertex against the world, view, and projection matrices.

output.position = mul(input.position, worldMatrix);

output.position = mul(output.position, viewMatrix);

output.position = mul(output.position, projectionMatrix);

// Store the texture coordinates for the pixel shader.

output.tex = input.tex;

return output;

}

Skyplane.ps

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplane.ps

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

The sky plane uses two cloud textures.

Texture2D cloudTexture1 : register(t0);

Texture2D cloudTexture2 : register(t1);

SamplerState SampleType;

The sky buffer contains the translation values for the two clouds as well as the cloud brightness value.

cbuffer SkyBuffer

{

float firstTranslationX;

float firstTranslationZ;

float secondTranslationX;

float secondTranslationZ;

float brightness;

float3 padding;

};

//////////////

// TYPEDEFS //

//////////////

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

};

The pixel shader is fairly simple. We sample the two clouds at their individual texture translation coordinates which creates the movement simulation. Then we combine the two cloud textures using a linear interpolation function. Finally we reduce the brightness of the clouds by the brightness value from the sky buffer.

////////////////////////////////////////////////////////////////////////////////

// Pixel Shader

////////////////////////////////////////////////////////////////////////////////

float4 SkyPlanePixelShader(PixelInputType input) : SV\_TARGET

{

float2 sampleLocation;

float4 textureColor1;

float4 textureColor2;

float4 finalColor;

// Translate the position where we sample the pixel from using the first texture translation values.

sampleLocation.x = input.tex.x + firstTranslationX;

sampleLocation.y = input.tex.y + firstTranslationZ;

// Sample the pixel color from the first cloud texture using the sampler at this texture coordinate location.

textureColor1 = cloudTexture1.Sample(SampleType, sampleLocation);

// Translate the position where we sample the pixel from using the second texture translation values.

sampleLocation.x = input.tex.x + secondTranslationX;

sampleLocation.y = input.tex.y + secondTranslationZ;

// Sample the pixel color from the second cloud texture using the sampler at this texture coordinate location.

textureColor2 = cloudTexture2.Sample(SampleType, sampleLocation);

// Combine the two cloud textures evenly.

finalColor = lerp(textureColor1, textureColor2, 0.5f);

// Reduce brightness of the combined cloud textures by the input brightness value.

finalColor = finalColor \* brightness;

return finalColor;

}

Skyplaneshaderclass.h

The SkyPlaneShaderClass is the shader used for rendering the clouds on the sky plane.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneshaderclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_SKYPLANESHADERCLASS\_H\_

#define \_SKYPLANESHADERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <d3dx11async.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: SkyPlaneShaderClass

////////////////////////////////////////////////////////////////////////////////

class SkyPlaneShaderClass

{

private:

struct MatrixBufferType

{

D3DXMATRIX world;

D3DXMATRIX view;

D3DXMATRIX projection;

};

The sky buffer type contains the translation coordinates for the clouds as well as the overall brightness of the clouds.

struct SkyBufferType

{

float firstTranslationX;

float firstTranslationZ;

float secondTranslationX;

float secondTranslationZ;

float brightness;

D3DXVECTOR3 padding;

};

public:

SkyPlaneShaderClass();

SkyPlaneShaderClass(const SkyPlaneShaderClass&);

~SkyPlaneShaderClass();

bool Initialize(ID3D11Device\*, HWND);

void Shutdown();

bool Render(ID3D11DeviceContext\*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*, float, float, float,

float, float);

private:

bool InitializeShader(ID3D11Device\*, HWND, WCHAR\*, WCHAR\*);

void ShutdownShader();

void OutputShaderErrorMessage(ID3D10Blob\*, HWND, WCHAR\*);

bool SetShaderParameters(ID3D11DeviceContext\*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*, float, float,

float, float, float);

void RenderShader(ID3D11DeviceContext\*, int);

private:

ID3D11VertexShader\* m\_vertexShader;

ID3D11PixelShader\* m\_pixelShader;

ID3D11InputLayout\* m\_layout;

ID3D11SamplerState\* m\_sampleState;

ID3D11Buffer\* m\_matrixBuffer;

We have a constant buffer that will be used for for setting the sky buffer information in the pixel shader.

ID3D11Buffer\* m\_skyBuffer;

};

#endif

Skyplaneshaderclass.h

The sky plane shader is the TextureShaderClass with modifications for rendering the sky plane.

////////////////////////////////////////////////////////////////////////////////

// Filename: skyplaneshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "skyplaneshaderclass.h"

SkyPlaneShaderClass::SkyPlaneShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

Initialize the sky constant buffer to null.

m\_skyBuffer = 0;

}

SkyPlaneShaderClass::SkyPlaneShaderClass(const SkyPlaneShaderClass& other)

{

}

SkyPlaneShaderClass::~SkyPlaneShaderClass()

{

}

bool SkyPlaneShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

Load the sky plane shaders.

// Initialize the vertex and pixel shaders.

result = InitializeShader(device, hwnd, L"../Engine/skyplane.vs", L"../Engine/skyplane.ps");

if(!result)

{

return false;

}

return true;

}

void SkyPlaneShaderClass::Shutdown()

{

// Shutdown the vertex and pixel shaders as well as the related objects.

ShutdownShader();

return;

}

bool SkyPlaneShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture, ID3D11ShaderResourceView\* texture2,

float firstTranslationX, float firstTranslationZ, float secondTranslationX, float secondTranslationZ, float brightness)

{

bool result;

// Set the shader parameters that it will use for rendering.

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture, texture2, firstTranslationX, firstTranslationZ,

secondTranslationX, secondTranslationZ, brightness);

if(!result)

{

return false;

}

// Now render the prepared buffers with the shader.

RenderShader(deviceContext, indexCount);

return true;

}

bool SkyPlaneShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[2];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC skyBufferDesc;

// Initialize the pointers this function will use to null.

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

Compile the sky plane vertex shader.

// Compile the vertex shader code.

result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "SkyPlaneVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&vertexShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

// If there was nothing in the error message then it simply could not find the shader file itself.

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

Compile the sky plane pixel shader.

// Compile the pixel shader code.

result = D3DX11CompileFromFile(psFilename, NULL, NULL, "SkyPlanePixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&pixelShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

// If there was nothing in the error message then it simply could not find the file itself.

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Create the vertex shader from the buffer.

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

// Create the pixel shader from the buffer.

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;

}

// Create the vertex input layout description.

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

// Get a count of the elements in the layout.

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

// Create the vertex input layout.

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;

}

// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

// Create a texture sampler state description.

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

// Create the texture sampler state.

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;

}

// Setup the description of the dynamic matrix constant buffer that is in the vertex shader.

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;

}

Setup a description of the sky constant buffer and then create it.

// Setup the description of the sky constant buffer that is in the pixel shader.

skyBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

skyBufferDesc.ByteWidth = sizeof(SkyBufferType);

skyBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

skyBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

skyBufferDesc.MiscFlags = 0;

skyBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the pixel shader constant buffer from within this class.

result = device->CreateBuffer(&skyBufferDesc, NULL, &m\_skyBuffer);

if(FAILED(result))

{

return false;

}

return true;

}

void SkyPlaneShaderClass::ShutdownShader()

{

Release the sky constant buffer in the ShutdownShader function.

// Release the sky constant buffer.

if(m\_skyBuffer)

{

m\_skyBuffer->Release();

m\_skyBuffer = 0;

}

// Release the matrix constant buffer.

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;

}

// Release the sampler states.

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;

}

// Release the layout.

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;

}

// Release the pixel shader.

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;

}

// Release the vertex shader.

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;

}

return;

}

void SkyPlaneShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

// Get a pointer to the error message text buffer.

compileErrors = (char\*)(errorMessage->GetBufferPointer());

// Get the length of the message.

bufferSize = errorMessage->GetBufferSize();

// Open a file to write the error message to.

fout.open("shader-error.txt");

// Write out the error message.

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

// Close the file.

fout.close();

// Release the error message.

errorMessage->Release();

errorMessage = 0;

// Pop a message up on the screen to notify the user to check the text file for compile errors.

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;

}

bool SkyPlaneShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView\* texture, ID3D11ShaderResourceView\* texture2,

float firstTranslationX, float firstTranslationZ, float secondTranslationX, float secondTranslationZ,

float brightness)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

MatrixBufferType\* dataPtr;

SkyBufferType\* dataPtr2;

unsigned int bufferNumber;

// Transpose the matrices to prepare them for the shader.

D3DXMatrixTranspose(&worldMatrix, &worldMatrix);

D3DXMatrixTranspose(&viewMatrix, &viewMatrix);

D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

// Lock the constant buffer so it can be written to.

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr = (MatrixBufferType\*)mappedResource.pData;

// Copy the matrices into the constant buffer.

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

// Unlock the constant buffer.

deviceContext->Unmap(m\_matrixBuffer, 0);

// Set the position of the constant buffer in the vertex shader.

bufferNumber = 0;

// Finally set the constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

Set the data contents in the sky constant buffer.

// Lock the sky constant buffer so it can be written to.

result = deviceContext->Map(m\_skyBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr2 = (SkyBufferType\*)mappedResource.pData;

// Copy the sky variables into the constant buffer.

dataPtr2->firstTranslationX = firstTranslationX;

dataPtr2->firstTranslationZ = firstTranslationZ;

dataPtr2->secondTranslationX = secondTranslationX;

dataPtr2->secondTranslationZ = secondTranslationZ;

dataPtr2->brightness = brightness;

dataPtr2->padding = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

// Unlock the constant buffer.

deviceContext->Unmap(m\_skyBuffer, 0);

// Set the position of the sky constant buffer in the pixel shader.

bufferNumber = 0;

// Finally set the sky constant buffer in the pixel shader with the updated values.

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_skyBuffer);

Set the two cloud textures in the pixel shader.

// Set the shader texture resource in the pixel shader.

deviceContext->PSSetShaderResources(0, 1, &texture);

deviceContext->PSSetShaderResources(1, 1, &texture2);

return true;

}

void SkyPlaneShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

// Set the vertex input layout.

deviceContext->IASetInputLayout(m\_layout);

// Set the vertex and pixel shaders that will be used to render the triangles.

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

// Set the sampler state in the pixel shader.

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

// Render the triangles.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}

D3dclass.h

The D3DClass has been modified for this tutorial by adding a secondary blend state for the cloud rendering.

////////////////////////////////////////////////////////////////////////////////

// Filename: d3dclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_D3DCLASS\_H\_

#define \_D3DCLASS\_H\_

/////////////

// LINKING //

/////////////

#pragma comment(lib, "dxgi.lib")

#pragma comment(lib, "d3d11.lib")

#pragma comment(lib, "d3dx11.lib")

#pragma comment(lib, "d3dx10.lib")

//////////////

// INCLUDES //

//////////////

#include <dxgi.h>

#include <d3dcommon.h>

#include <d3d11.h>

#include <d3dx10math.h>

////////////////////////////////////////////////////////////////////////////////

// Class name: D3DClass

////////////////////////////////////////////////////////////////////////////////

class D3DClass

{

public:

D3DClass();

D3DClass(const D3DClass&);

~D3DClass();

bool Initialize(int, int, bool, HWND, bool, float, float);

void Shutdown();

void BeginScene(float, float, float, float);

void EndScene();

ID3D11Device\* GetDevice();

ID3D11DeviceContext\* GetDeviceContext();

void GetProjectionMatrix(D3DXMATRIX&);

void GetWorldMatrix(D3DXMATRIX&);

void GetOrthoMatrix(D3DXMATRIX&);

void GetVideoCardInfo(char\*, int&);

void TurnZBufferOn();

void TurnZBufferOff();

void TurnOnAlphaBlending();

void TurnOffAlphaBlending();

void TurnOnCulling();

void TurnOffCulling();

This is the new function for enabling the additive blending that the clouds will require.

void EnableSecondBlendState();

private:

bool m\_vsync\_enabled;

int m\_videoCardMemory;

char m\_videoCardDescription[128];

IDXGISwapChain\* m\_swapChain;

ID3D11Device\* m\_device;

ID3D11DeviceContext\* m\_deviceContext;

ID3D11RenderTargetView\* m\_renderTargetView;

ID3D11Texture2D\* m\_depthStencilBuffer;

ID3D11DepthStencilState\* m\_depthStencilState;

ID3D11DepthStencilView\* m\_depthStencilView;

ID3D11RasterizerState\* m\_rasterState;

ID3D11RasterizerState\* m\_rasterStateNoCulling;

D3DXMATRIX m\_projectionMatrix;

D3DXMATRIX m\_worldMatrix;

D3DXMATRIX m\_orthoMatrix;

ID3D11DepthStencilState\* m\_depthDisabledStencilState;

ID3D11BlendState\* m\_alphaEnableBlendingState;

ID3D11BlendState\* m\_alphaDisableBlendingState;

This is the new blend state pointer.

ID3D11BlendState\* m\_alphaBlendState2;

};

#endif

D3dclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: d3dclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "d3dclass.h"

D3DClass::D3DClass()

{

m\_swapChain = 0;

m\_device = 0;

m\_deviceContext = 0;

m\_renderTargetView = 0;

m\_depthStencilBuffer = 0;

m\_depthStencilState = 0;

m\_depthStencilView = 0;

m\_rasterState = 0;

m\_rasterStateNoCulling = 0;

m\_depthDisabledStencilState = 0;

m\_alphaEnableBlendingState = 0;

m\_alphaDisableBlendingState = 0;

Initialize the new blend state pointer to null in the class constructor.

m\_alphaBlendState2 = 0;

}

D3DClass::D3DClass(const D3DClass& other)

{

}

D3DClass::~D3DClass()

{

}

bool D3DClass::Initialize(int screenWidth, int screenHeight, bool vsync, HWND hwnd, bool fullscreen, float screenDepth, float screenNear)

{

HRESULT result;

IDXGIFactory\* factory;

IDXGIAdapter\* adapter;

IDXGIOutput\* adapterOutput;

unsigned int numModes, i, numerator, denominator, stringLength;

DXGI\_MODE\_DESC\* displayModeList;

DXGI\_ADAPTER\_DESC adapterDesc;

int error;

DXGI\_SWAP\_CHAIN\_DESC swapChainDesc;

D3D\_FEATURE\_LEVEL featureLevel;

ID3D11Texture2D\* backBufferPtr;

D3D11\_TEXTURE2D\_DESC depthBufferDesc;

D3D11\_DEPTH\_STENCIL\_DESC depthStencilDesc;

D3D11\_DEPTH\_STENCIL\_VIEW\_DESC depthStencilViewDesc;

D3D11\_RASTERIZER\_DESC rasterDesc;

D3D11\_VIEWPORT viewport;

float fieldOfView, screenAspect;

D3D11\_DEPTH\_STENCIL\_DESC depthDisabledStencilDesc;

D3D11\_BLEND\_DESC blendStateDescription;

// Store the vsync setting.

m\_vsync\_enabled = vsync;

// Create a DirectX graphics interface factory.

result = CreateDXGIFactory(\_\_uuidof(IDXGIFactory), (void\*\*)&factory);

if(FAILED(result))

{

return false;

}

// Use the factory to create an adapter for the primary graphics interface (video card).

result = factory->EnumAdapters(0, &adapter);

if(FAILED(result))

{

return false;

}

// Enumerate the primary adapter output (monitor).

result = adapter->EnumOutputs(0, &adapterOutput);

if(FAILED(result))

{

return false;

}

// Get the number of modes that fit the DXGI\_FORMAT\_R8G8B8A8\_UNORM display format for the adapter output (monitor).

result = adapterOutput->GetDisplayModeList(DXGI\_FORMAT\_R8G8B8A8\_UNORM, DXGI\_ENUM\_MODES\_INTERLACED, &numModes, NULL);

if(FAILED(result))

{

return false;

}

// Create a list to hold all the possible display modes for this monitor/video card combination.

displayModeList = new DXGI\_MODE\_DESC[numModes];

if(!displayModeList)

{

return false;

}

// Now fill the display mode list structures.

result = adapterOutput->GetDisplayModeList(DXGI\_FORMAT\_R8G8B8A8\_UNORM, DXGI\_ENUM\_MODES\_INTERLACED, &numModes, displayModeList);

if(FAILED(result))

{

return false;

}

// Now go through all the display modes and find the one that matches the screen width and height.

// When a match is found store the numerator and denominator of the refresh rate for that monitor.

for(i=0; i<numModes; i++)

{

if(displayModeList[i].Width == (unsigned int)screenWidth)

{

if(displayModeList[i].Height == (unsigned int)screenHeight)

{

numerator = displayModeList[i].RefreshRate.Numerator;

denominator = displayModeList[i].RefreshRate.Denominator;

}

}

}

// Get the adapter (video card) description.

result = adapter->GetDesc(&adapterDesc);

if(FAILED(result))

{

return false;

}

// Store the dedicated video card memory in megabytes.

m\_videoCardMemory = (int)(adapterDesc.DedicatedVideoMemory / 1024 / 1024);

// Convert the name of the video card to a character array and store it.

error = wcstombs\_s(&stringLength, m\_videoCardDescription, 128, adapterDesc.Description, 128);

if(error != 0)

{

return false;

}

// Release the display mode list.

delete [] displayModeList;

displayModeList = 0;

// Release the adapter output.

adapterOutput->Release();

adapterOutput = 0;

// Release the adapter.

adapter->Release();

adapter = 0;

// Release the factory.

factory->Release();

factory = 0;

// Initialize the swap chain description.

ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));

// Set to a single back buffer.

swapChainDesc.BufferCount = 1;

// Set the width and height of the back buffer.

swapChainDesc.BufferDesc.Width = screenWidth;

swapChainDesc.BufferDesc.Height = screenHeight;

// Set regular 32-bit surface for the back buffer.

swapChainDesc.BufferDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

// Set the refresh rate of the back buffer.

if(m\_vsync\_enabled)

{

swapChainDesc.BufferDesc.RefreshRate.Numerator = numerator;

swapChainDesc.BufferDesc.RefreshRate.Denominator = denominator;

}

else

{

swapChainDesc.BufferDesc.RefreshRate.Numerator = 0;

swapChainDesc.BufferDesc.RefreshRate.Denominator = 1;

}

// Set the usage of the back buffer.

swapChainDesc.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

// Set the handle for the window to render to.

swapChainDesc.OutputWindow = hwnd;

// Turn multisampling off.

swapChainDesc.SampleDesc.Count = 1;

swapChainDesc.SampleDesc.Quality = 0;

// Set to full screen or windowed mode.

if(fullscreen)

{

swapChainDesc.Windowed = false;

}

else

{

swapChainDesc.Windowed = true;

}

// Set the scan line ordering and scaling to unspecified.

swapChainDesc.BufferDesc.ScanlineOrdering = DXGI\_MODE\_SCANLINE\_ORDER\_UNSPECIFIED;

swapChainDesc.BufferDesc.Scaling = DXGI\_MODE\_SCALING\_UNSPECIFIED;

// Discard the back buffer contents after presenting.

swapChainDesc.SwapEffect = DXGI\_SWAP\_EFFECT\_DISCARD;

// Don't set the advanced flags.

swapChainDesc.Flags = 0;

// Set the feature level to DirectX 11.

featureLevel = D3D\_FEATURE\_LEVEL\_11\_0;

// Create the swap chain, Direct3D device, and Direct3D device context.

result = D3D11CreateDeviceAndSwapChain(NULL, D3D\_DRIVER\_TYPE\_HARDWARE, NULL, 0, &featureLevel, 1,

D3D11\_SDK\_VERSION, &swapChainDesc, &m\_swapChain, &m\_device, NULL, &m\_deviceContext);

if(FAILED(result))

{

return false;

}

// Get the pointer to the back buffer.

result = m\_swapChain->GetBuffer(0, \_\_uuidof(ID3D11Texture2D), (LPVOID\*)&backBufferPtr);

if(FAILED(result))

{

return false;

}

// Create the render target view with the back buffer pointer.

result = m\_device->CreateRenderTargetView(backBufferPtr, NULL, &m\_renderTargetView);

if(FAILED(result))

{

return false;

}

// Release pointer to the back buffer as we no longer need it.

backBufferPtr->Release();

backBufferPtr = 0;

// Initialize the description of the depth buffer.

ZeroMemory(&depthBufferDesc, sizeof(depthBufferDesc));

// Set up the description of the depth buffer.

depthBufferDesc.Width = screenWidth;

depthBufferDesc.Height = screenHeight;

depthBufferDesc.MipLevels = 1;

depthBufferDesc.ArraySize = 1;

depthBufferDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

depthBufferDesc.SampleDesc.Count = 1;

depthBufferDesc.SampleDesc.Quality = 0;

depthBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

depthBufferDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

depthBufferDesc.CPUAccessFlags = 0;

depthBufferDesc.MiscFlags = 0;

// Create the texture for the depth buffer using the filled out description.

result = m\_device->CreateTexture2D(&depthBufferDesc, NULL, &m\_depthStencilBuffer);

if(FAILED(result))

{

return false;

}

// Initialize the description of the stencil state.

ZeroMemory(&depthStencilDesc, sizeof(depthStencilDesc));

// Set up the description of the stencil state.

depthStencilDesc.DepthEnable = true;

depthStencilDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;

depthStencilDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

depthStencilDesc.StencilEnable = true;

depthStencilDesc.StencilReadMask = 0xFF;

depthStencilDesc.StencilWriteMask = 0xFF;

// Stencil operations if pixel is front-facing.

depthStencilDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INCR;

depthStencilDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

// Stencil operations if pixel is back-facing.

depthStencilDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_DECR;

depthStencilDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthStencilDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

// Create the depth stencil state.

result = m\_device->CreateDepthStencilState(&depthStencilDesc, &m\_depthStencilState);

if(FAILED(result))

{

return false;

}

// Set the depth stencil state.

m\_deviceContext->OMSetDepthStencilState(m\_depthStencilState, 1);

// Initialize the depth stencil view.

ZeroMemory(&depthStencilViewDesc, sizeof(depthStencilViewDesc));

// Set up the depth stencil view description.

depthStencilViewDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

depthStencilViewDesc.ViewDimension = D3D11\_DSV\_DIMENSION\_TEXTURE2D;

depthStencilViewDesc.Texture2D.MipSlice = 0;

// Create the depth stencil view.

result = m\_device->CreateDepthStencilView(m\_depthStencilBuffer, &depthStencilViewDesc, &m\_depthStencilView);

if(FAILED(result))

{

return false;

}

// Bind the render target view and depth stencil buffer to the output render pipeline.

m\_deviceContext->OMSetRenderTargets(1, &m\_renderTargetView, m\_depthStencilView);

// Setup the raster description which will determine how and what polygons will be drawn.

rasterDesc.AntialiasedLineEnable = false;

rasterDesc.CullMode = D3D11\_CULL\_BACK;

rasterDesc.DepthBias = 0;

rasterDesc.DepthBiasClamp = 0.0f;

rasterDesc.DepthClipEnable = true;

rasterDesc.FillMode = D3D11\_FILL\_SOLID;

rasterDesc.FrontCounterClockwise = false;

rasterDesc.MultisampleEnable = false;

rasterDesc.ScissorEnable = false;

rasterDesc.SlopeScaledDepthBias = 0.0f;

// Create the rasterizer state from the description we just filled out.

result = m\_device->CreateRasterizerState(&rasterDesc, &m\_rasterState);

if(FAILED(result))

{

return false;

}

// Now set the rasterizer state.

m\_deviceContext->RSSetState(m\_rasterState);

// Setup a raster description which turns off back face culling.

rasterDesc.AntialiasedLineEnable = false;

rasterDesc.CullMode = D3D11\_CULL\_NONE;

rasterDesc.DepthBias = 0;

rasterDesc.DepthBiasClamp = 0.0f;

rasterDesc.DepthClipEnable = true;

rasterDesc.FillMode = D3D11\_FILL\_SOLID;

rasterDesc.FrontCounterClockwise = false;

rasterDesc.MultisampleEnable = false;

rasterDesc.ScissorEnable = false;

rasterDesc.SlopeScaledDepthBias = 0.0f;

// Create the no culling rasterizer state.

result = m\_device->CreateRasterizerState(&rasterDesc, &m\_rasterStateNoCulling);

if(FAILED(result))

{

return false;

}

// Setup the viewport for rendering.

viewport.Width = (float)screenWidth;

viewport.Height = (float)screenHeight;

viewport.MinDepth = 0.0f;

viewport.MaxDepth = 1.0f;

viewport.TopLeftX = 0.0f;

viewport.TopLeftY = 0.0f;

// Create the viewport.

m\_deviceContext->RSSetViewports(1, &viewport);

// Setup the projection matrix.

fieldOfView = (float)D3DX\_PI / 4.0f;

screenAspect = (float)screenWidth / (float)screenHeight;

// Create the projection matrix for 3D rendering.

D3DXMatrixPerspectiveFovLH(&m\_projectionMatrix, fieldOfView, screenAspect, screenNear, screenDepth);

// Initialize the world matrix to the identity matrix.

D3DXMatrixIdentity(&m\_worldMatrix);

// Create an orthographic projection matrix for 2D rendering.

D3DXMatrixOrthoLH(&m\_orthoMatrix, (float)screenWidth, (float)screenHeight, screenNear, screenDepth);

// Clear the second depth stencil state before setting the parameters.

ZeroMemory(&depthDisabledStencilDesc, sizeof(depthDisabledStencilDesc));

// Now create a second depth stencil state which turns off the Z buffer for 2D rendering. The only difference is

// that DepthEnable is set to false, all other parameters are the same as the other depth stencil state.

depthDisabledStencilDesc.DepthEnable = false;

depthDisabledStencilDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;

depthDisabledStencilDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

depthDisabledStencilDesc.StencilEnable = true;

depthDisabledStencilDesc.StencilReadMask = 0xFF;

depthDisabledStencilDesc.StencilWriteMask = 0xFF;

depthDisabledStencilDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_INCR;

depthDisabledStencilDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

depthDisabledStencilDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_DECR;

depthDisabledStencilDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_KEEP;

depthDisabledStencilDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

// Create the state using the device.

result = m\_device->CreateDepthStencilState(&depthDisabledStencilDesc, &m\_depthDisabledStencilState);

if(FAILED(result))

{

return false;

}

// Clear the blend state description.

ZeroMemory(&blendStateDescription, sizeof(D3D11\_BLEND\_DESC));

// Create an alpha enabled blend state description.

blendStateDescription.RenderTarget[0].BlendEnable = TRUE;

blendStateDescription.RenderTarget[0].SrcBlend = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].DestBlend = D3D11\_BLEND\_INV\_SRC\_ALPHA;

blendStateDescription.RenderTarget[0].BlendOp = D3D11\_BLEND\_OP\_ADD;

blendStateDescription.RenderTarget[0].SrcBlendAlpha = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].DestBlendAlpha = D3D11\_BLEND\_ZERO;

blendStateDescription.RenderTarget[0].BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

blendStateDescription.RenderTarget[0].RenderTargetWriteMask = 0x0f;

// Create the blend state using the description.

result = m\_device->CreateBlendState(&blendStateDescription, &m\_alphaEnableBlendingState);

if(FAILED(result))

{

return false;

}

// Modify the description to create an alpha disabled blend state description.

blendStateDescription.RenderTarget[0].BlendEnable = FALSE;

// Create the second blend state using the description.

result = m\_device->CreateBlendState(&blendStateDescription, &m\_alphaDisableBlendingState);

if(FAILED(result))

{

return false;

}

Here we setup the description of the additive blend state and then create it.

// Create a secondary alpha blend state description.

blendStateDescription.RenderTarget[0].BlendEnable = TRUE;

blendStateDescription.RenderTarget[0].SrcBlend = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].DestBlend = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].BlendOp = D3D11\_BLEND\_OP\_ADD;

blendStateDescription.RenderTarget[0].SrcBlendAlpha = D3D11\_BLEND\_ONE;

blendStateDescription.RenderTarget[0].DestBlendAlpha = D3D11\_BLEND\_ZERO;

blendStateDescription.RenderTarget[0].BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

blendStateDescription.RenderTarget[0].RenderTargetWriteMask = 0x0f;

// Create the blend state using the description.

result = m\_device->CreateBlendState(&blendStateDescription, &m\_alphaBlendState2);

if(FAILED(result))

{

return false;

}

return true;

}

void D3DClass::Shutdown()

{

// Before shutting down set to windowed mode or when you release the swap chain it will throw an exception.

if(m\_swapChain)

{

m\_swapChain->SetFullscreenState(false, NULL);

}

Release the new blend state in the Shutdown function.

if(m\_alphaBlendState2)

{

m\_alphaBlendState2->Release();

m\_alphaBlendState2 = 0;

}

if(m\_alphaEnableBlendingState)

{

m\_alphaEnableBlendingState->Release();

m\_alphaEnableBlendingState = 0;

}

if(m\_alphaDisableBlendingState)

{

m\_alphaDisableBlendingState->Release();

m\_alphaDisableBlendingState = 0;

}

if(m\_depthDisabledStencilState)

{

m\_depthDisabledStencilState->Release();

m\_depthDisabledStencilState = 0;

}

if(m\_rasterStateNoCulling)

{

m\_rasterStateNoCulling->Release();

m\_rasterStateNoCulling = 0;

}

if(m\_rasterState)

{

m\_rasterState->Release();

m\_rasterState = 0;

}

if(m\_depthStencilView)

{

m\_depthStencilView->Release();

m\_depthStencilView = 0;

}

if(m\_depthStencilState)

{

m\_depthStencilState->Release();

m\_depthStencilState = 0;

}

if(m\_depthStencilBuffer)

{

m\_depthStencilBuffer->Release();

m\_depthStencilBuffer = 0;

}

if(m\_renderTargetView)

{

m\_renderTargetView->Release();

m\_renderTargetView = 0;

}

if(m\_deviceContext)

{

m\_deviceContext->Release();

m\_deviceContext = 0;

}

if(m\_device)

{

m\_device->Release();

m\_device = 0;

}

if(m\_swapChain)

{

m\_swapChain->Release();

m\_swapChain = 0;

}

return;

}

void D3DClass::BeginScene(float red, float green, float blue, float alpha)

{

float color[4];

// Setup the color to clear the buffer to.

color[0] = red;

color[1] = green;

color[2] = blue;

color[3] = alpha;

// Clear the back buffer.

m\_deviceContext->ClearRenderTargetView(m\_renderTargetView, color);

// Clear the depth buffer.

m\_deviceContext->ClearDepthStencilView(m\_depthStencilView, D3D11\_CLEAR\_DEPTH, 1.0f, 0);

return;

}

void D3DClass::EndScene()

{

// Present the back buffer to the screen since rendering is complete.

if(m\_vsync\_enabled)

{

// Lock to screen refresh rate.

m\_swapChain->Present(1, 0);

}

else

{

// Present as fast as possible.

m\_swapChain->Present(0, 0);

}

return;

}

ID3D11Device\* D3DClass::GetDevice()

{

return m\_device;

}

ID3D11DeviceContext\* D3DClass::GetDeviceContext()

{

return m\_deviceContext;

}

void D3DClass::GetProjectionMatrix(D3DXMATRIX& projectionMatrix)

{

projectionMatrix = m\_projectionMatrix;

return;

}

void D3DClass::GetWorldMatrix(D3DXMATRIX& worldMatrix)

{

worldMatrix = m\_worldMatrix;

return;

}

void D3DClass::GetOrthoMatrix(D3DXMATRIX& orthoMatrix)

{

orthoMatrix = m\_orthoMatrix;

return;

}

void D3DClass::GetVideoCardInfo(char\* cardName, int& memory)

{

strcpy\_s(cardName, 128, m\_videoCardDescription);

memory = m\_videoCardMemory;

return;

}

void D3DClass::TurnZBufferOn()

{

m\_deviceContext->OMSetDepthStencilState(m\_depthStencilState, 1);

return;

}

void D3DClass::TurnZBufferOff()

{

m\_deviceContext->OMSetDepthStencilState(m\_depthDisabledStencilState, 1);

return;

}

void D3DClass::TurnOnAlphaBlending()

{

float blendFactor[4];

// Setup the blend factor.

blendFactor[0] = 0.0f;

blendFactor[1] = 0.0f;

blendFactor[2] = 0.0f;

blendFactor[3] = 0.0f;

// Turn on the alpha blending.

m\_deviceContext->OMSetBlendState(m\_alphaEnableBlendingState, blendFactor, 0xffffffff);

return;

}

void D3DClass::TurnOffAlphaBlending()

{

float blendFactor[4];

// Setup the blend factor.

blendFactor[0] = 0.0f;

blendFactor[1] = 0.0f;

blendFactor[2] = 0.0f;

blendFactor[3] = 0.0f;

// Turn off the alpha blending.

m\_deviceContext->OMSetBlendState(m\_alphaDisableBlendingState, blendFactor, 0xffffffff);

return;

}

void D3DClass::TurnOnCulling()

{

// Set the culling rasterizer state.

m\_deviceContext->RSSetState(m\_rasterState);

return;

}

void D3DClass::TurnOffCulling()

{

// Set the no back face culling rasterizer state.

m\_deviceContext->RSSetState(m\_rasterStateNoCulling);

return;

}

This is the new function for enabling the additive blend state.

void D3DClass::EnableSecondBlendState()

{

float blendFactor[4];

// Setup the blend factor.

blendFactor[0] = 0.0f;

blendFactor[1] = 0.0f;

blendFactor[2] = 0.0f;

blendFactor[3] = 0.0f;

// Turn on the alpha blending.

m\_deviceContext->OMSetBlendState(m\_alphaBlendState2, blendFactor, 0xffffffff);

return;

}

Applicationclass.h

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_APPLICATIONCLASS\_H\_

#define \_APPLICATIONCLASS\_H\_

/////////////

// GLOBALS //

/////////////

const bool FULL\_SCREEN = true;

const bool VSYNC\_ENABLED = true;

const float SCREEN\_DEPTH = 1000.0f;

const float SCREEN\_NEAR = 0.1f;

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "inputclass.h"

#include "d3dclass.h"

#include "cameraclass.h"

#include "terrainclass.h"

#include "timerclass.h"

#include "positionclass.h"

#include "fpsclass.h"

#include "cpuclass.h"

#include "fontshaderclass.h"

#include "textclass.h"

#include "terrainshaderclass.h"

#include "lightclass.h"

#include "skydomeclass.h"

#include "skydomeshaderclass.h"

The new SkyPlaneClass and SkyPlaneShaderClass headers are included here in the ApplicationClass.

#include "skyplaneclass.h"

#include "skyplaneshaderclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: ApplicationClass

////////////////////////////////////////////////////////////////////////////////

class ApplicationClass

{

public:

ApplicationClass();

ApplicationClass(const ApplicationClass&);

~ApplicationClass();

bool Initialize(HINSTANCE, HWND, int, int);

void Shutdown();

bool Frame();

private:

bool HandleInput(float);

bool RenderGraphics();

private:

InputClass\* m\_Input;

D3DClass\* m\_Direct3D;

CameraClass\* m\_Camera;

TerrainClass\* m\_Terrain;

TimerClass\* m\_Timer;

PositionClass\* m\_Position;

FpsClass\* m\_Fps;

CpuClass\* m\_Cpu;

FontShaderClass\* m\_FontShader;

TextClass\* m\_Text;

TerrainShaderClass\* m\_TerrainShader;

LightClass\* m\_Light;

SkyDomeClass\* m\_SkyDome;

SkyDomeShaderClass\* m\_SkyDomeShader;

The pointers to the SkyPlaneClass and SkyPlaneShaderClass are added here.

SkyPlaneClass \*m\_SkyPlane;

SkyPlaneShaderClass\* m\_SkyPlaneShader;

};

#endif

Applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "applicationclass.h"

ApplicationClass::ApplicationClass()

{

m\_Input = 0;

m\_Direct3D = 0;

m\_Camera = 0;

m\_Terrain = 0;

m\_Timer = 0;

m\_Position = 0;

m\_Fps = 0;

m\_Cpu = 0;

m\_FontShader = 0;

m\_Text = 0;

m\_TerrainShader = 0;

m\_Light = 0;

m\_SkyDome = 0;

m\_SkyDomeShader = 0;

The pointers to the SkyPlaneClass and SkyPlaneShaderClass are initialized to null in the class constructor.

m\_SkyPlane = 0;

m\_SkyPlaneShader = 0;

}

ApplicationClass::ApplicationClass(const ApplicationClass& other)

{

}

ApplicationClass::~ApplicationClass()

{

}

bool ApplicationClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)

{

bool result;

float cameraX, cameraY, cameraZ;

D3DXMATRIX baseViewMatrix;

char videoCard[128];

int videoMemory;

// Create the input object. The input object will be used to handle reading the keyboard and mouse input from the user.

m\_Input = new InputClass;

if(!m\_Input)

{

return false;

}

// Initialize the input object.

result = m\_Input->Initialize(hinstance, hwnd, screenWidth, screenHeight);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the input object.", L"Error", MB\_OK);

return false;

}

// Create the Direct3D object.

m\_Direct3D = new D3DClass;

if(!m\_Direct3D)

{

return false;

}

// Initialize the Direct3D object.

result = m\_Direct3D->Initialize(screenWidth, screenHeight, VSYNC\_ENABLED, hwnd, FULL\_SCREEN, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize DirectX 11.", L"Error", MB\_OK);

return false;

}

// Create the camera object.

m\_Camera = new CameraClass;

if(!m\_Camera)

{

return false;

}

// Initialize a base view matrix with the camera for 2D user interface rendering.

m\_Camera->SetPosition(0.0f, 0.0f, -1.0f);

m\_Camera->Render();

m\_Camera->GetViewMatrix(baseViewMatrix);

// Set the initial position of the camera.

cameraX = 50.0f;

cameraY = 2.0f;

cameraZ = -7.0f;

m\_Camera->SetPosition(cameraX, cameraY, cameraZ);

// Create the terrain object.

m\_Terrain = new TerrainClass;

if(!m\_Terrain)

{

return false;

}

// Initialize the terrain object.

result = m\_Terrain->Initialize(m\_Direct3D->GetDevice(), "../Engine/data/heightmap01.bmp", L"../Engine/data/dirt01.dds", "../Engine/data/colorm01.bmp");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain object.", L"Error", MB\_OK);

return false;

}

// Create the timer object.

m\_Timer = new TimerClass;

if(!m\_Timer)

{

return false;

}

// Initialize the timer object.

result = m\_Timer->Initialize();

if(!result)

{

MessageBox(hwnd, L"Could not initialize the timer object.", L"Error", MB\_OK);

return false;

}

// Create the position object.

m\_Position = new PositionClass;

if(!m\_Position)

{

return false;

}

// Set the initial position of the viewer to the same as the initial camera position.

m\_Position->SetPosition(cameraX, cameraY, cameraZ);

// Create the fps object.

m\_Fps = new FpsClass;

if(!m\_Fps)

{

return false;

}

// Initialize the fps object.

m\_Fps->Initialize();

// Create the cpu object.

m\_Cpu = new CpuClass;

if(!m\_Cpu)

{

return false;

}

// Initialize the cpu object.

m\_Cpu->Initialize();

// Create the font shader object.

m\_FontShader = new FontShaderClass;

if(!m\_FontShader)

{

return false;

}

// Initialize the font shader object.

result = m\_FontShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the font shader object.", L"Error", MB\_OK);

return false;

}

// Create the text object.

m\_Text = new TextClass;

if(!m\_Text)

{

return false;

}

// Initialize the text object.

result = m\_Text->Initialize(m\_Direct3D->GetDevice(), m\_Direct3D->GetDeviceContext(), hwnd, screenWidth, screenHeight, baseViewMatrix);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the text object.", L"Error", MB\_OK);

return false;

}

// Retrieve the video card information.

m\_Direct3D->GetVideoCardInfo(videoCard, videoMemory);

// Set the video card information in the text object.

result = m\_Text->SetVideoCardInfo(videoCard, videoMemory, m\_Direct3D->GetDeviceContext());

if(!result)

{

MessageBox(hwnd, L"Could not set video card info in the text object.", L"Error", MB\_OK);

return false;

}

// Create the terrain shader object.

m\_TerrainShader = new TerrainShaderClass;

if(!m\_TerrainShader)

{

return false;

}

// Initialize the terrain shader object.

result = m\_TerrainShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain shader object.", L"Error", MB\_OK);

return false;

}

// Create the light object.

m\_Light = new LightClass;

if(!m\_Light)

{

return false;

}

// Initialize the light object.

m\_Light->SetAmbientColor(0.05f, 0.05f, 0.05f, 1.0f);

m\_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);

m\_Light->SetDirection(-0.5f, -1.0f, 0.0f);

// Create the sky dome object.

m\_SkyDome = new SkyDomeClass;

if(!m\_SkyDome)

{

return false;

}

// Initialize the sky dome object.

result = m\_SkyDome->Initialize(m\_Direct3D->GetDevice());

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome object.", L"Error", MB\_OK);

return false;

}

// Create the sky dome shader object.

m\_SkyDomeShader = new SkyDomeShaderClass;

if(!m\_SkyDomeShader)

{

return false;

}

// Initialize the sky dome shader object.

result = m\_SkyDomeShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome shader object.", L"Error", MB\_OK);

return false;

}

The new sky plane and the sky plane shader are created and initialized here.

// Create the sky plane object.

m\_SkyPlane = new SkyPlaneClass;

if(!m\_SkyPlane)

{

return false;

}

// Initialize the sky plane object.

result = m\_SkyPlane->Initialize(m\_Direct3D->GetDevice(), L"../Engine/data/cloud001.dds", L"../Engine/data/cloud002.dds");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky plane object.", L"Error", MB\_OK);

return false;

}

// Create the sky plane shader object.

m\_SkyPlaneShader = new SkyPlaneShaderClass;

if(!m\_SkyPlaneShader)

{

return false;

}

// Initialize the sky plane shader object.

result = m\_SkyPlaneShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky plane shader object.", L"Error", MB\_OK);

return false;

}

return true;

}

void ApplicationClass::Shutdown()

{

The sky plane and the sky plane shader are released here in the Shutdown function.

// Release the sky plane shader object.

if(m\_SkyPlaneShader)

{

m\_SkyPlaneShader->Shutdown();

delete m\_SkyPlaneShader;

m\_SkyPlaneShader = 0;

}

// Release the sky plane object.

if(m\_SkyPlane)

{

m\_SkyPlane->Shutdown();

delete m\_SkyPlane;

m\_SkyPlane = 0;

}

// Release the sky dome shader object.

if(m\_SkyDomeShader)

{

m\_SkyDomeShader->Shutdown();

delete m\_SkyDomeShader;

m\_SkyDomeShader = 0;

}

// Release the sky dome object.

if(m\_SkyDome)

{

m\_SkyDome->Shutdown();

delete m\_SkyDome;

m\_SkyDome = 0;

}

// Release the light object.

if(m\_Light)

{

delete m\_Light;

m\_Light = 0;

}

// Release the terrain shader object.

if(m\_TerrainShader)

{

m\_TerrainShader->Shutdown();

delete m\_TerrainShader;

m\_TerrainShader = 0;

}

// Release the text object.

if(m\_Text)

{

m\_Text->Shutdown();

delete m\_Text;

m\_Text = 0;

}

// Release the font shader object.

if(m\_FontShader)

{

m\_FontShader->Shutdown();

delete m\_FontShader;

m\_FontShader = 0;

}

// Release the cpu object.

if(m\_Cpu)

{

m\_Cpu->Shutdown();

delete m\_Cpu;

m\_Cpu = 0;

}

// Release the fps object.

if(m\_Fps)

{

delete m\_Fps;

m\_Fps = 0;

}

// Release the position object.

if(m\_Position)

{

delete m\_Position;

m\_Position = 0;

}

// Release the timer object.

if(m\_Timer)

{

delete m\_Timer;

m\_Timer = 0;

}

// Release the terrain object.

if(m\_Terrain)

{

m\_Terrain->Shutdown();

delete m\_Terrain;

m\_Terrain = 0;

}

// Release the camera object.

if(m\_Camera)

{

delete m\_Camera;

m\_Camera = 0;

}

// Release the Direct3D object.

if(m\_Direct3D)

{

m\_Direct3D->Shutdown();

delete m\_Direct3D;

m\_Direct3D = 0;

}

// Release the input object.

if(m\_Input)

{

m\_Input->Shutdown();

delete m\_Input;

m\_Input = 0;

}

return;

}

bool ApplicationClass::Frame()

{

bool result;

// Read the user input.

result = m\_Input->Frame();

if(!result)

{

return false;

}

// Check if the user pressed escape and wants to exit the application.

if(m\_Input->IsEscapePressed() == true)

{

return false;

}

// Update the system stats.

m\_Timer->Frame();

m\_Fps->Frame();

m\_Cpu->Frame();

// Update the FPS value in the text object.

result = m\_Text->SetFps(m\_Fps->GetFps(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the CPU usage value in the text object.

result = m\_Text->SetCpu(m\_Cpu->GetCpuPercentage(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Do the frame input processing.

result = HandleInput(m\_Timer->GetTime());

if(!result)

{

return false;

}

The sky plane needs to update the texture translation each frame. Note that if you are going to unlock the vsync you should pass the frame time in here.

// Do the sky plane frame processing.

m\_SkyPlane->Frame();

// Render the graphics.

result = RenderGraphics();

if(!result)

{

return false;

}

return result;

}

bool ApplicationClass::HandleInput(float frameTime)

{

bool keyDown, result;

float posX, posY, posZ, rotX, rotY, rotZ;

// Set the frame time for calculating the updated position.

m\_Position->SetFrameTime(frameTime);

// Handle the input.

keyDown = m\_Input->IsLeftPressed();

m\_Position->TurnLeft(keyDown);

keyDown = m\_Input->IsRightPressed();

m\_Position->TurnRight(keyDown);

keyDown = m\_Input->IsUpPressed();

m\_Position->MoveForward(keyDown);

keyDown = m\_Input->IsDownPressed();

m\_Position->MoveBackward(keyDown);

keyDown = m\_Input->IsAPressed();

m\_Position->MoveUpward(keyDown);

keyDown = m\_Input->IsZPressed();

m\_Position->MoveDownward(keyDown);

keyDown = m\_Input->IsPgUpPressed();

m\_Position->LookUpward(keyDown);

keyDown = m\_Input->IsPgDownPressed();

m\_Position->LookDownward(keyDown);

// Get the view point position/rotation.

m\_Position->GetPosition(posX, posY, posZ);

m\_Position->GetRotation(rotX, rotY, rotZ);

// Set the position of the camera.

m\_Camera->SetPosition(posX, posY, posZ);

m\_Camera->SetRotation(rotX, rotY, rotZ);

// Update the position values in the text object.

result = m\_Text->SetCameraPosition(posX, posY, posZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the rotation values in the text object.

result = m\_Text->SetCameraRotation(rotX, rotY, rotZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

return true;

}

bool ApplicationClass::RenderGraphics()

{

D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix;

D3DXVECTOR3 cameraPosition;

bool result;

// Clear the scene.

m\_Direct3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

// Generate the view matrix based on the camera's position.

m\_Camera->Render();

// Get the world, view, projection, and ortho matrices from the camera and Direct3D objects.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

m\_Direct3D->GetOrthoMatrix(orthoMatrix);

// Get the position of the camera.

cameraPosition = m\_Camera->GetPosition();

// Translate the sky dome to be centered around the camera position.

D3DXMatrixTranslation(&worldMatrix, cameraPosition.x, cameraPosition.y, cameraPosition.z);

// Turn off back face culling.

m\_Direct3D->TurnOffCulling();

// Turn off the Z buffer.

m\_Direct3D->TurnZBufferOff();

// Render the sky dome using the sky dome shader.

m\_SkyDome->Render(m\_Direct3D->GetDeviceContext());

m\_SkyDomeShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyDome->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyDome->GetApexColor(), m\_SkyDome->GetCenterColor());

// Turn back face culling back on.

m\_Direct3D->TurnOnCulling();

Turn on the additive blending and then render the sky plane here.

// Enable additive blending so the clouds blend with the sky dome color.

m\_Direct3D->EnableSecondBlendState();

// Render the sky plane using the sky plane shader.

m\_SkyPlane->Render(m\_Direct3D->GetDeviceContext());

m\_SkyPlaneShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyPlane->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyPlane->GetCloudTexture1(), m\_SkyPlane->GetCloudTexture2(), m\_SkyPlane->GetTranslation(0), m\_SkyPlane->GetTranslation(1),

m\_SkyPlane->GetTranslation(2), m\_SkyPlane->GetTranslation(3), m\_SkyPlane->GetBrightness());

// Turn off blending.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on.

m\_Direct3D->TurnZBufferOn();

// Reset the world matrix.

m\_Direct3D->GetWorldMatrix(worldMatrix);

// Render the terrain using the terrain shader.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

result = m\_TerrainShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_Light->GetAmbientColor(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(), m\_Terrain->GetTexture());

if(!result)

{

return false;

}

// Turn off the Z buffer to begin all 2D rendering.

m\_Direct3D->TurnZBufferOff();

// Turn on the alpha blending before rendering the text.

m\_Direct3D->TurnOnAlphaBlending();

// Render the text user interface elements.

result = m\_Text->Render(m\_Direct3D->GetDeviceContext(), m\_FontShader, worldMatrix, orthoMatrix);

if(!result)

{

return false;

}

// Turn off alpha blending after rendering the text.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on now that all 2D rendering has completed.

m\_Direct3D->TurnZBufferOn();

// Present the rendered scene to the screen.

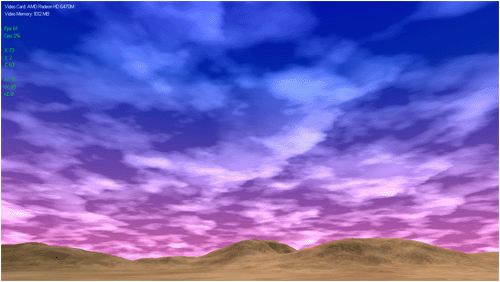
m\_Direct3D->EndScene();

return true;

}

Summary

We now have a sky plane with clouds rendered on it that blend with the color dome creating a fairly realistic cloud effect. The two layers of clouds moving at different speeds gives us the illusion of depth as well.



To Do Exercises

1. Compile and run the program. You should see two layers of clouds moving. Use PgUp to look up at the sky.

2. Play around with the parameters SkyPlaneClass::Initialize to see the effects they have on the sky plane.

3. Use some different cloud textures to generate you own desired sky appearance.

4. Setup varying wind conditions over time to modify the direction and speed of your clouds as time proceeds.

5. Create another input variable to do a subtraction in the pixel shader on the cloud texture. This will have the effect of shrinking the clouds which can add more variation to the cloud scene.

Source Code

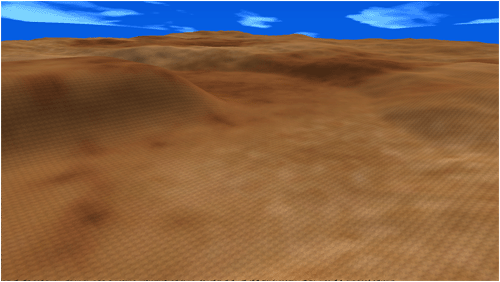
**Tutorial 16: Small Body Water**

http://web.archive.org/web/20140722200827/http:/rastertek.com/pic1001.gif

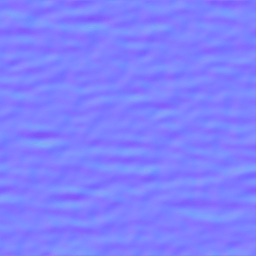
This DirectX 11 terrain tutorial will cover how to implement small body water using HLSL and C++. The code in this tutorial builds on the previous terrain tutorials.

To implement small body water (lakes, rivers, ponds, swamps, and anything else that doesn't have big waves) we use a refractive and reflective rendering system. We render a refraction (everything under the water) and render a reflection (everything above the water) and then combine the results. If you have gone over the water tutorial in the DirectX tutorial section then you will understand these concepts. If you haven't read that tutorial then I suggest you review it first before proceeding as we simply expand on it here and add more features.

We will begin with a basic terrain scene. We require terrain for the refraction and something such as sky for the reflection.



We will also need a normal map for the water ripple effect. A gaussian blurred height map and the Nvidia normal map filter for Photoshop can create something like the following:



We then create a single quad with a radius large enough to cover the area of the water. We will be tiling the normal map texture over the quad so we don't need it subdivided.

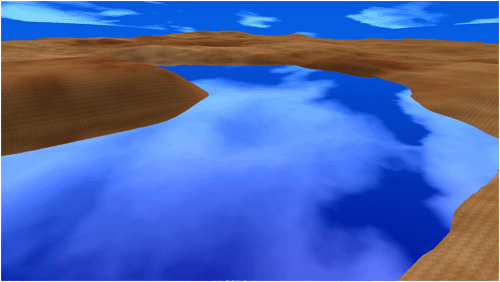
Next we will render the scene using the reflection shader but use an inverted clip plane so we are rendering everything underneath the water to create a refraction. This is all rendered to a render to texture so we can then apply it to the water quad using projective texturing based on the view point of the camera. We also perturb the refraction texture sampling by the water normal map to give us the ripple effect.

As well when we tile the normal map over the water quad we do it two separate times using different texture coordinates. For example we may tile it ten times over the water and then tile it again but just five times. We then combine the two separately tiled normal map results to give an animated look to the ripples instead of just a single rotating texture.

And finally we add a tint color to the refraction so that it looks like colored water instead of purely clear water with ripples. The following image is what the water refraction ends up looking like:

Note that a lot of engines will stop at that point. It is very expensive to render the scene to texture since we basically are rendering the scene twice. This cuts the fps in half whenever water needs to be drawn! There aren't really a lot of techniques other than to not render to texture when the water isn't visible that can speed this up unfortunately.

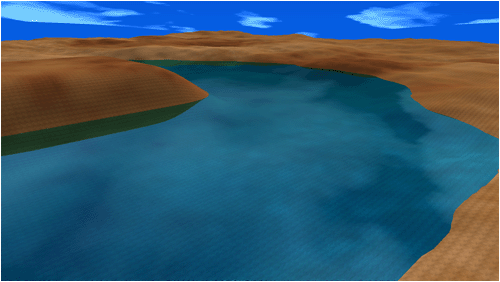
However we are going to proceed to make the effect look better. We will now render the scene yet again using the reflection shader but using a clip plane that renders everything above the water to give us a reflection of our scene. We render the reflection to a render to texture object and then project it onto the water quad from the view point of the camera. We also perturb the sampling of the render to texture by the water normal map. Rendering just the reflection looks like the following:



With the refraction, reflection, and the regular 3D scene we have now ended up drawing the scene three times per frame. This cuts your fps down to one third of the original speed. Some optimizations are that you can render a reflection twice a second instead of every frame, but it depends on the maximum speed that you will ever move at over the water. Some engines also use just a single texture of a sky that never changes. There are different things you can do but it depends on how important the reflection is to the scene.

With both the refraction and the reflection rendered to textures we can now combine them. One of the nice ways to combine the two textures is to use what is known as the Fresnel factor. This is the effect that as you get closer to the water it becomes more transparent (refraction is rendered more), and as you get further away the reflection becomes stronger and it is no longer as easy to see into the water (reflection is rendered more). In this tutorial I have created a simple fresnel factor that is based only on height. So the higher up you are the water is more reflective, and the closer you get to the water it becomes more refractive. Some engines will use a far more sophisticated frensel calculation and take into account the distance from the water, the depth of the water at each pixel, and so forth. It just depends how realistic you want to make the effect look.

The combined result with the fresnel factor looks like the following:



As you can see the reflection added a lot of color to the result as well as the reflected clouds and pieces of the terrain.

The final addition to the water effect is adding specular reflection for where the directional sun light reflects off the water. We will calculate the specular against the dual tiled water normal map so that the specular comes off just the animated water ripples. Doing so gives us the final resulting scene:

Once again this is a fairly expensive effect, you may want to just stop at the refraction point. But if you can spare the extra cycles then the full effect is nice.

To begin the code section we will start by looking at the modified CameraClass.

Cameraclass.h

The CameraClass has been modified so that it can now generate reflection view matrices.

////////////////////////////////////////////////////////////////////////////////

// Filename: cameraclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_CAMERACLASS\_H\_

#define \_CAMERACLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3dx10math.h>

////////////////////////////////////////////////////////////////////////////////

// Class name: CameraClass

////////////////////////////////////////////////////////////////////////////////

class CameraClass

{

public:

CameraClass();

CameraClass(const CameraClass&);

~CameraClass();

void SetPosition(float, float, float);

void SetRotation(float, float, float);

D3DXVECTOR3 GetPosition();

D3DXVECTOR3 GetRotation();

void Render();

void GetViewMatrix(D3DXMATRIX&);

void GenerateBaseViewMatrix();

void GetBaseViewMatrix(D3DXMATRIX&);

We have added functions to generate and retrieve the reflection view matrix.

void RenderReflection(float);

void GetReflectionViewMatrix(D3DXMATRIX&);

private:

float m\_positionX, m\_positionY, m\_positionZ;

float m\_rotationX, m\_rotationY, m\_rotationZ;

There is also an additional matrix for the reflection view.

D3DXMATRIX m\_viewMatrix, m\_baseViewMatrix, m\_reflectionViewMatrix;

};

#endif

Cameraclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: cameraclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "cameraclass.h"

CameraClass::CameraClass()

{

m\_positionX = 0.0f;

m\_positionY = 0.0f;

m\_positionZ = 0.0f;

m\_rotationX = 0.0f;

m\_rotationY = 0.0f;

m\_rotationZ = 0.0f;

}

CameraClass::CameraClass(const CameraClass& other)

{

}

CameraClass::~CameraClass()

{

}

void CameraClass::SetPosition(float x, float y, float z)

{

m\_positionX = x;

m\_positionY = y;

m\_positionZ = z;

return;

}

void CameraClass::SetRotation(float x, float y, float z)

{

m\_rotationX = x;

m\_rotationY = y;

m\_rotationZ = z;

return;

}

D3DXVECTOR3 CameraClass::GetPosition()

{

return D3DXVECTOR3(m\_positionX, m\_positionY, m\_positionZ);

}

D3DXVECTOR3 CameraClass::GetRotation()

{

return D3DXVECTOR3(m\_rotationX, m\_rotationY, m\_rotationZ);

}

void CameraClass::Render()

{

D3DXVECTOR3 up, position, lookAt;

float yaw, pitch, roll;

D3DXMATRIX rotationMatrix;

// Setup the vector that points upwards.

up.x = 0.0f;

up.y = 1.0f;

up.z = 0.0f;

// Setup the position of the camera in the world.

position.x = m\_positionX;

position.y = m\_positionY;

position.z = m\_positionZ;

// Setup where the camera is looking by default.

lookAt.x = 0.0f;

lookAt.y = 0.0f;

lookAt.z = 1.0f;

// Set the yaw (Y axis), pitch (X axis), and roll (Z axis) rotations in radians.

pitch = m\_rotationX \* 0.0174532925f;

yaw = m\_rotationY \* 0.0174532925f;

roll = m\_rotationZ \* 0.0174532925f;

// Create the rotation matrix from the yaw, pitch, and roll values.

D3DXMatrixRotationYawPitchRoll(&rotationMatrix, yaw, pitch, roll);

// Transform the lookAt and up vector by the rotation matrix so the view is correctly rotated at the origin.

D3DXVec3TransformCoord(&lookAt, &lookAt, &rotationMatrix);

D3DXVec3TransformCoord(&up, &up, &rotationMatrix);

// Translate the rotated camera position to the location of the viewer.

lookAt = position + lookAt;

// Finally create the view matrix from the three updated vectors.

D3DXMatrixLookAtLH(&m\_viewMatrix, &position, &lookAt, &up);

return;

}

void CameraClass::GetViewMatrix(D3DXMATRIX& viewMatrix)

{

viewMatrix = m\_viewMatrix;

return;

}

void CameraClass::GenerateBaseViewMatrix()

{

D3DXVECTOR3 up, position, lookAt;

float yaw, pitch, roll;

D3DXMATRIX rotationMatrix;

// Setup the vector that points upwards.

up.x = 0.0f;

up.y = 1.0f;

up.z = 0.0f;

// Setup the position of the camera in the world.

position.x = m\_positionX;

position.y = m\_positionY;

position.z = m\_positionZ;

// Setup where the camera is looking by default.

lookAt.x = 0.0f;

lookAt.y = 0.0f;

lookAt.z = 1.0f;

// Set the yaw (Y axis), pitch (X axis), and roll (Z axis) rotations in radians.

pitch = m\_rotationX \* 0.0174532925f;

yaw = m\_rotationY \* 0.0174532925f;

roll = m\_rotationZ \* 0.0174532925f;

// Create the rotation matrix from the yaw, pitch, and roll values.

D3DXMatrixRotationYawPitchRoll(&rotationMatrix, yaw, pitch, roll);

// Transform the lookAt and up vector by the rotation matrix so the view is correctly rotated at the origin.

D3DXVec3TransformCoord(&lookAt, &lookAt, &rotationMatrix);

D3DXVec3TransformCoord(&up, &up, &rotationMatrix);

// Translate the rotated camera position to the location of the viewer.

lookAt = position + lookAt;

// Finally create the view matrix from the three updated vectors.

D3DXMatrixLookAtLH(&m\_baseViewMatrix, &position, &lookAt, &up);

return;

}

void CameraClass::GetBaseViewMatrix(D3DXMATRIX& viewMatrix)

{

viewMatrix = m\_baseViewMatrix;

return;

}

This is the new function that is used for generating the reflection view matrix. The only difference between the regular view matrix and the reflection one is that we invert the Y position based on the height of the plane, and we also invert the pitch.

void CameraClass::RenderReflection(float height)

{

D3DXVECTOR3 up, position, lookAt;

float yaw, pitch, roll;

D3DXMATRIX rotationMatrix;

// Setup the vector that points upwards.

up.x = 0.0f;

up.y = 1.0f;

up.z = 0.0f;

// Setup the position of the camera in the world. For planar reflection invert the Y position of the camera.

position.x = m\_positionX;

position.y = -m\_positionY + (height \* 2.0f);

position.z = m\_positionZ;

// Setup where the camera is looking by default.

lookAt.x = 0.0f;

lookAt.y = 0.0f;

lookAt.z = 1.0f;

// Set the yaw (Y axis), pitch (X axis), and roll (Z axis) rotations in radians. Invert the X rotation for reflection.

pitch = -m\_rotationX \* 0.0174532925f;

yaw = m\_rotationY \* 0.0174532925f;

roll = m\_rotationZ \* 0.0174532925f;

// Create the rotation matrix from the yaw, pitch, and roll values.

D3DXMatrixRotationYawPitchRoll(&rotationMatrix, yaw, pitch, roll);

// Transform the lookAt and up vector by the rotation matrix so the view is correctly rotated at the origin.

D3DXVec3TransformCoord(&lookAt, &lookAt, &rotationMatrix);

D3DXVec3TransformCoord(&up, &up, &rotationMatrix);

// Translate the rotated camera position to the location of the viewer.

lookAt = position + lookAt;

// Finally create the reflection view matrix from the three updated vectors.

D3DXMatrixLookAtLH(&m\_reflectionViewMatrix, &position, &lookAt, &up);

return;

}

There is also a new function to retrieve the reflection view matrix.

void CameraClass::GetReflectionViewMatrix(D3DXMATRIX& viewMatrix)

{

viewMatrix = m\_reflectionViewMatrix;

return;

}

Reflection.vs

The reflection HLSL shaders are just the terrain HLSL shaders with a clip plane added, otherwise they are identical.

////////////////////////////////////////////////////////////////////////////////

// Filename: reflection.vs

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer MatrixBuffer

{

matrix worldMatrix;

matrix viewMatrix;

matrix projectionMatrix;

};

cbuffer ClipPlaneBuffer

{

float4 clipPlane;

};

//////////////

// TYPEDEFS //

//////////////

struct VertexInputType

{

float4 position : POSITION;

float2 tex : TEXCOORD0;

float3 normal : NORMAL;

float3 tangent : TANGENT;

float3 binormal : BINORMAL;

float4 color : COLOR;

};

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

float3 normal : NORMAL;

float3 tangent : TANGENT;

float3 binormal : BINORMAL;

float4 color : COLOR;

float clip : SV\_ClipDistance0;

};

////////////////////////////////////////////////////////////////////////////////

// Vertex Shader

////////////////////////////////////////////////////////////////////////////////

PixelInputType ReflectionVertexShader(VertexInputType input)

{

PixelInputType output;

// Change the position vector to be 4 units for proper matrix calculations.

input.position.w = 1.0f;

// Calculate the position of the vertex against the world, view, and projection matrices.

output.position = mul(input.position, worldMatrix);

output.position = mul(output.position, viewMatrix);

output.position = mul(output.position, projectionMatrix);

// Store the texture coordinates for the pixel shader.

output.tex = input.tex;

// Calculate the normal vector against the world matrix only and then normalize the final value.

output.normal = mul(input.normal, (float3x3)worldMatrix);

output.normal = normalize(output.normal);

// Calculate the tangent vector against the world matrix only and then normalize the final value.

output.tangent = mul(input.tangent, (float3x3)worldMatrix);

output.tangent = normalize(output.tangent);

// Calculate the binormal vector against the world matrix only and then normalize the final value.

output.binormal = mul(input.binormal, (float3x3)worldMatrix);

output.binormal = normalize(output.binormal);

// Send the color map color into the pixel shader.

output.color = input.color;

// Set the clipping plane.

output.clip = dot(mul(input.position, worldMatrix), clipPlane);

return output;

}

Reflection.ps

////////////////////////////////////////////////////////////////////////////////

// Filename: reflection.ps

////////////////////////////////////////////////////////////////////////////////

//////////////

// TEXTURES //

//////////////

Texture2D colorTexture : register(t0);

Texture2D normalTexture : register(t1);

//////////////

// SAMPLERS //

//////////////

SamplerState SampleType;

//////////////////////

// CONSTANT BUFFERS //

//////////////////////

cbuffer LightBuffer

{

float4 lightDiffuseColor;

float3 lightDirection;

float colorTextureBrightness;

};

//////////////

// TYPEDEFS //

//////////////

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

float3 normal : NORMAL;

float3 tangent : TANGENT;

float3 binormal : BINORMAL;

float4 color : COLOR;

float clip : SV\_ClipDistance0;

};

////////////////////////////////////////////////////////////////////////////////

// Pixel Shader

////////////////////////////////////////////////////////////////////////////////

float4 ReflectionPixelShader(PixelInputType input) : SV\_TARGET

{

float3 lightDir;

float4 textureColor;

float4 bumpMap;

float3 bumpNormal;

float lightIntensity;

float4 color;

// Invert the light direction for calculations.

lightDir = -lightDirection;

// Sample the color texture.

textureColor = colorTexture.Sample(SampleType, input.tex);

// Combine the color map value into the texture color.

textureColor = saturate(input.color \* textureColor \* colorTextureBrightness);

// Calculate the bump map using the normal map.

bumpMap = normalTexture.Sample(SampleType, input.tex);

bumpMap = (bumpMap \* 2.0f) - 1.0f;

bumpNormal = input.normal + bumpMap.x \* input.tangent + bumpMap.y \* input.binormal;

bumpNormal = normalize(bumpNormal);

lightIntensity = saturate(dot(bumpNormal, lightDir));

// Calculate the first bump mapped pixel color.

color = saturate(lightDiffuseColor \* lightIntensity);

color = color \* textureColor;

return color;

}

Reflectionshaderclass.h

The ReflectionShaderClass is also the same as the TerrainShaderClass except that it has clip plane related settings.

////////////////////////////////////////////////////////////////////////////////

// Filename: reflectionshaderclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_REFLECTIONSHADERCLASS\_H\_

#define \_REFLECTIONSHADERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <d3dx11async.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: ReflectionShaderClass

////////////////////////////////////////////////////////////////////////////////

class ReflectionShaderClass

{

private:

struct MatrixBufferType

{

D3DXMATRIX world;

D3DXMATRIX view;

D3DXMATRIX projection;

};

struct ClipPlaneBufferType

{

D3DXVECTOR4 clipPlane;

};

struct LightBufferType

{

D3DXVECTOR4 lightDiffuseColor;

D3DXVECTOR3 lightDirection;

float colorTextureBrightness;

};

public:

ReflectionShaderClass();

ReflectionShaderClass(const ReflectionShaderClass&);

~ReflectionShaderClass();

bool Initialize(ID3D11Device\*, HWND);

void Shutdown();

bool Render(ID3D11DeviceContext\*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*, D3DXVECTOR4,

D3DXVECTOR3, float, D3DXVECTOR4);

private:

bool InitializeShader(ID3D11Device\*, HWND, WCHAR\*, WCHAR\*);

void ShutdownShader();

void OutputShaderErrorMessage(ID3D10Blob\*, HWND, WCHAR\*);

bool SetShaderParameters(ID3D11DeviceContext\*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*, D3DXVECTOR4,

D3DXVECTOR3, float, D3DXVECTOR4);

void RenderShader(ID3D11DeviceContext\*, int);

private:

ID3D11VertexShader\* m\_vertexShader;

ID3D11PixelShader\* m\_pixelShader;

ID3D11InputLayout\* m\_layout;

ID3D11SamplerState\* m\_sampleState;

ID3D11Buffer\* m\_matrixBuffer;

ID3D11Buffer\* m\_clipPlaneBuffer;

ID3D11Buffer\* m\_lightBuffer;

};

#endif

Reflectionshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: reflectionshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "reflectionshaderclass.h"

ReflectionShaderClass::ReflectionShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

m\_clipPlaneBuffer = 0;

m\_lightBuffer = 0;

}

ReflectionShaderClass::ReflectionShaderClass(const ReflectionShaderClass& other)

{

}

ReflectionShaderClass::~ReflectionShaderClass()

{

}

bool ReflectionShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

// Initialize the vertex and pixel shaders.

result = InitializeShader(device, hwnd, L"../Engine/reflection.vs", L"../Engine/reflection.ps");

if(!result)

{

return false;

}

return true;

}

void ReflectionShaderClass::Shutdown()

{

// Shutdown the vertex and pixel shaders as well as the related objects.

ShutdownShader();

return;

}

bool ReflectionShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView\* colorTexture, ID3D11ShaderResourceView\* normalTexture,

D3DXVECTOR4 lightDiffuseColor, D3DXVECTOR3 lightDirection, float colorTextureBrightness, D3DXVECTOR4 clipPlane)

{

bool result;

// Set the shader parameters that it will use for rendering.

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, colorTexture, normalTexture, lightDiffuseColor,

lightDirection, colorTextureBrightness, clipPlane);

if(!result)

{

return false;

}

// Now render the prepared buffers with the shader.

RenderShader(deviceContext, indexCount);

return true;

}

bool ReflectionShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[6];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC clipPlaneBufferDesc;

D3D11\_BUFFER\_DESC lightBufferDesc;

// Initialize the pointers this function will use to null.

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

// Compile the vertex shader code.

result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "ReflectionVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&vertexShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

// If there was nothing in the error message then it simply could not find the shader file itself.

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Compile the pixel shader code.

result = D3DX11CompileFromFile(psFilename, NULL, NULL, "ReflectionPixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&pixelShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

// If there was nothing in the error message then it simply could not find the file itself.

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Create the vertex shader from the buffer.

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

// Create the pixel shader from the buffer.

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;

}

// Create the vertex input layout description.

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

polygonLayout[2].SemanticName = "NORMAL";

polygonLayout[2].SemanticIndex = 0;

polygonLayout[2].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[2].InputSlot = 0;

polygonLayout[2].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[2].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[2].InstanceDataStepRate = 0;

polygonLayout[3].SemanticName = "TANGENT";

polygonLayout[3].SemanticIndex = 0;

polygonLayout[3].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[3].InputSlot = 0;

polygonLayout[3].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[3].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[3].InstanceDataStepRate = 0;

polygonLayout[4].SemanticName = "BINORMAL";

polygonLayout[4].SemanticIndex = 0;

polygonLayout[4].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[4].InputSlot = 0;

polygonLayout[4].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[4].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[4].InstanceDataStepRate = 0;

polygonLayout[5].SemanticName = "COLOR";

polygonLayout[5].SemanticIndex = 0;

polygonLayout[5].Format = DXGI\_FORMAT\_R32G32B32A32\_FLOAT;

polygonLayout[5].InputSlot = 0;

polygonLayout[5].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[5].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[5].InstanceDataStepRate = 0;

// Get a count of the elements in the layout.

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

// Create the vertex input layout.

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;

}

// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

// Create a texture sampler state description.

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_CLAMP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_CLAMP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_CLAMP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

// Create the texture sampler state.

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;

}

// Setup the description of the dynamic matrix constant buffer that is in the vertex shader.

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;

}

// Setup the description of the clip plane dynamic constant buffer that is in the vertex shader.

clipPlaneBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

clipPlaneBufferDesc.ByteWidth = sizeof(ClipPlaneBufferType);

clipPlaneBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

clipPlaneBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

clipPlaneBufferDesc.MiscFlags = 0;

clipPlaneBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&clipPlaneBufferDesc, NULL, &m\_clipPlaneBuffer);

if(FAILED(result))

{

return false;

}

// Setup the description of the light dynamic constant buffer that is in the pixel shader.

lightBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

lightBufferDesc.ByteWidth = sizeof(LightBufferType);

lightBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

lightBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

lightBufferDesc.MiscFlags = 0;

lightBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the pixel shader constant buffer from within this class.

result = device->CreateBuffer(&lightBufferDesc, NULL, &m\_lightBuffer);

if(FAILED(result))

{

return false;

}

return true;

}

void ReflectionShaderClass::ShutdownShader()

{

// Release the light constant buffer.

if(m\_lightBuffer)

{

m\_lightBuffer->Release();

m\_lightBuffer = 0;

}

// Release the clip plane constant buffer.

if(m\_clipPlaneBuffer)

{

m\_clipPlaneBuffer->Release();

m\_clipPlaneBuffer = 0;

}

// Release the matrix constant buffer.

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;

}

// Release the sampler state.

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;

}

// Release the layout.

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;

}

// Release the pixel shader.

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;

}

// Release the vertex shader.

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;

}

return;

}

void ReflectionShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

// Get a pointer to the error message text buffer.

compileErrors = (char\*)(errorMessage->GetBufferPointer());

// Get the length of the message.

bufferSize = errorMessage->GetBufferSize();

// Open a file to write the error message to.

fout.open("shader-error.txt");

// Write out the error message.

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

// Close the file.

fout.close();

// Release the error message.

errorMessage->Release();

errorMessage = 0;

// Pop a message up on the screen to notify the user to check the text file for compile errors.

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;

}

bool ReflectionShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView\* colorTexture,

ID3D11ShaderResourceView\* normalTexture, D3DXVECTOR4 lightDiffuseColor, D3DXVECTOR3 lightDirection,

float colorTextureBrightness, D3DXVECTOR4 clipPlane)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

unsigned int bufferNumber;

MatrixBufferType\* dataPtr;

ClipPlaneBufferType\* dataPtr1;

LightBufferType\* dataPtr2;

// Transpose the matrices to prepare them for the shader.

D3DXMatrixTranspose(&worldMatrix, &worldMatrix);

D3DXMatrixTranspose(&viewMatrix, &viewMatrix);

D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

// Lock the constant buffer so it can be written to.

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr = (MatrixBufferType\*)mappedResource.pData;

// Copy the matrices into the constant buffer.

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

// Unlock the constant buffer.

deviceContext->Unmap(m\_matrixBuffer, 0);

// Set the position of the constant buffer in the vertex shader.

bufferNumber = 0;

// Now set the constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

// Lock the clip plane constant buffer so it can be written to.

result = deviceContext->Map(m\_clipPlaneBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the clip plane constant buffer.

dataPtr1 = (ClipPlaneBufferType\*)mappedResource.pData;

// Copy the clip plane into the clip plane constant buffer.

dataPtr1->clipPlane = clipPlane;

// Unlock the buffer.

deviceContext->Unmap(m\_clipPlaneBuffer, 0);

// Set the position of the clip plane constant buffer in the vertex shader.

bufferNumber = 1;

// Now set the clip plane constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_clipPlaneBuffer);

// Lock the light constant buffer so it can be written to.

result = deviceContext->Map(m\_lightBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr2 = (LightBufferType\*)mappedResource.pData;

// Copy the lighting variables into the constant buffer.

dataPtr2->lightDiffuseColor = lightDiffuseColor;

dataPtr2->lightDirection = lightDirection;

dataPtr2->colorTextureBrightness = colorTextureBrightness;

// Unlock the constant buffer.

deviceContext->Unmap(m\_lightBuffer, 0);

// Set the position of the light constant buffer in the pixel shader.

bufferNumber = 0;

// Finally set the light constant buffer in the pixel shader with the updated values.

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_lightBuffer);

// Set the texture resources in the pixel shader.

deviceContext->PSSetShaderResources(0, 1, &colorTexture);

deviceContext->PSSetShaderResources(1, 1, &normalTexture);

return true;

}

void ReflectionShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

// Set the vertex input layout.

deviceContext->IASetInputLayout(m\_layout);

// Set the vertex and pixel shaders that will be used to render this triangle.

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

// Set the sampler state in the pixel shader.

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

// Render the triangle.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}

Waterclass.h

WaterClass is a new class that is used to hold the geometry for the water quad and the shader related settings and files.

////////////////////////////////////////////////////////////////////////////////

// Filename: waterclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_WATERCLASS\_H\_

#define \_WATERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "textureclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: WaterClass

////////////////////////////////////////////////////////////////////////////////

class WaterClass

{

private:

struct VertexType

{

D3DXVECTOR3 position;

D3DXVECTOR2 texture;

};

public:

WaterClass();

WaterClass(const WaterClass&);

~WaterClass();

bool Initialize(ID3D11Device\*, WCHAR\*, float, float);

void Shutdown();

void Frame();

void Render(ID3D11DeviceContext\*);

int GetIndexCount();

ID3D11ShaderResourceView\* GetTexture();

float GetWaterHeight();

D3DXVECTOR2 GetNormalMapTiling();

float GetWaterTranslation();

float GetReflectRefractScale();

D3DXVECTOR4 GetRefractionTint();

float GetSpecularShininess();

private:

bool InitializeBuffers(ID3D11Device\*, float);

void ShutdownBuffers();

void RenderBuffers(ID3D11DeviceContext\*);

bool LoadTexture(ID3D11Device\*, WCHAR\*);

void ReleaseTexture();

private:

float m\_waterHeight;

ID3D11Buffer \*m\_vertexBuffer, \*m\_indexBuffer;

int m\_vertexCount, m\_indexCount;

TextureClass\* m\_Texture;

D3DXVECTOR2 m\_normalMapTiling;

float m\_waterTranslation;

float m\_reflectRefractScale;

D3DXVECTOR4 m\_refractionTint;

float m\_specularShininess;

};

#endif

Waterclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: waterclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "waterclass.h"

WaterClass::WaterClass()

{

m\_vertexBuffer = 0;

m\_indexBuffer = 0;

m\_Texture = 0;

}

WaterClass::WaterClass(const WaterClass& other)

{

}

WaterClass::~WaterClass()

{

}

The Initialize function takes as inputs the DX11 device, the file name of the water normal map, the height of the water, and the radius of the water.

bool WaterClass::Initialize(ID3D11Device\* device, WCHAR\* textureFilename, float waterHeight, float waterRadius)

{

bool result;

First we store the water height. After that we create the geometry for the water quad and then load the normal map for the water.

// Store the water height.

m\_waterHeight = waterHeight;

// Initialize the vertex and index buffer that hold the geometry for the triangle.

result = InitializeBuffers(device, waterRadius);

if(!result)

{

return false;

}

// Load the texture for this model.

result = LoadTexture(device, textureFilename);

if(!result)

{

return false;

}

Next we set all the shader related settings. These should actually be input parameters but I have placed them here for now so that they are straight forward for editing as you learn how each of them affects the shader effect.

// Set the tiling for the water normal maps.

m\_normalMapTiling.x = 0.01f; // Tile ten times over the quad.

m\_normalMapTiling.y = 0.02f; // Tile five times over the quad.

// Initialize the water translation to zero.

m\_waterTranslation = 0.0f;

// Set the scaling value for the water normal map.

m\_reflectRefractScale = 0.03f;

// Set the tint of the refraction.

m\_refractionTint = D3DXVECTOR4(0.0f, 0.8f, 1.0f, 1.0f);

// Set the specular shininess.

m\_specularShininess = 200.0f;

return true;

}

void WaterClass::Shutdown()

{

// Release the model texture.

ReleaseTexture();

// Release the vertex and index buffers.

ShutdownBuffers();

return;

}

Each frame we will animate the rotation of the water normal map to simulate moving water ripples.

void WaterClass::Frame()

{

// Update the position of the water to simulate motion.

m\_waterTranslation += 0.003f;

if(m\_waterTranslation > 1.0f)

{

m\_waterTranslation -= 1.0f;

}

return;

}

void WaterClass::Render(ID3D11DeviceContext\* deviceContext)

{

// Put the vertex and index buffers on the graphics pipeline to prepare them for drawing.

RenderBuffers(deviceContext);

return;

}

int WaterClass::GetIndexCount()

{

return m\_indexCount;

}

ID3D11ShaderResourceView\* WaterClass::GetTexture()

{

return m\_Texture->GetTexture();

}

float WaterClass::GetWaterHeight()

{

return m\_waterHeight;

}

D3DXVECTOR2 WaterClass::GetNormalMapTiling()

{

return m\_normalMapTiling;

}

float WaterClass::GetWaterTranslation()

{

return m\_waterTranslation;

}

float WaterClass::GetReflectRefractScale()

{

return m\_reflectRefractScale;

}

D3DXVECTOR4 WaterClass::GetRefractionTint()

{

return m\_refractionTint;

}

float WaterClass::GetSpecularShininess()

{

return m\_specularShininess;

}

bool WaterClass::InitializeBuffers(ID3D11Device\* device, float waterRadius)

{

VertexType\* vertices;

unsigned long\* indices;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

// Set the number of vertices in the vertex array.

m\_vertexCount = 6;

// Set the number of indices in the index array.

m\_indexCount = 6;

// Create the vertex array.

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;

}

// Create the index array.

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;

}

We manually create a quad here. It doesn't need to be a high poly object since we are going to tile the water normal map over the surface of the quad.

// Load the vertex array with data.

vertices[0].position = D3DXVECTOR3(-waterRadius, 0.0f, waterRadius); // Top left.

vertices[0].texture = D3DXVECTOR2(0.0f, 0.0f);

vertices[1].position = D3DXVECTOR3(waterRadius, 0.0f, waterRadius); // Top right.

vertices[1].texture = D3DXVECTOR2(1.0f, 0.0f);

vertices[2].position = D3DXVECTOR3(-waterRadius, 0.0f, -waterRadius); // Bottom left.

vertices[2].texture = D3DXVECTOR2(0.0f, 1.0f);

vertices[3].position = D3DXVECTOR3(-waterRadius, 0.0f, -waterRadius); // Bottom left.

vertices[3].texture = D3DXVECTOR2(0.0f, 1.0f);

vertices[4].position = D3DXVECTOR3(waterRadius, 0.0f, waterRadius); // Top right.

vertices[4].texture = D3DXVECTOR2(1.0f, 0.0f);

vertices[5].position = D3DXVECTOR3(waterRadius, 0.0f, -waterRadius); // Bottom right.

vertices[5].texture = D3DXVECTOR2(1.0f, 1.0f);

// Load the index array with data.

indices[0] = 0;

indices[1] = 1;

indices[2] = 2;

indices[3] = 3;

indices[4] = 4;

indices[5] = 5;

// Set up the description of the vertex buffer.

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the vertex data.

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

// Now finally create the vertex buffer.

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{

return false;

}

// Set up the description of the index buffer.

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the index data.

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

// Create the index buffer.

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{

return false;

}

// Release the arrays now that the vertex and index buffers have been created and loaded.

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;

}

void WaterClass::ShutdownBuffers()

{

// Release the index buffer.

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;

}

// Release the vertex buffer.

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;

}

return;

}

void WaterClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

// Set vertex buffer stride and offset.

stride = sizeof(VertexType);

offset = 0;

// Set the vertex buffer to active in the input assembler so it can be rendered.

deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

// Set the index buffer to active in the input assembler so it can be rendered.

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}

The texture that is loaded here is the normal map for the water.

bool WaterClass::LoadTexture(ID3D11Device\* device, WCHAR\* filename)

{

bool result;

// Create the texture object.

m\_Texture = new TextureClass;

if(!m\_Texture)

{

return false;

}

// Initialize the texture object.

result = m\_Texture->Initialize(device, filename);

if(!result)

{

return false;

}

return true;

}

void WaterClass::ReleaseTexture()

{

// Release the texture object.

if(m\_Texture)

{

m\_Texture->Shutdown();

delete m\_Texture;

m\_Texture = 0;

}

return;

}

Water.vs

The HLSL water shaders here are very similar to the one in the DirectX tutorials section. All we have done is add some additional effects.

////////////////////////////////////////////////////////////////////////////////

// Filename: water.vs

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer MatrixBuffer

{

matrix worldMatrix;

matrix viewMatrix;

matrix projectionMatrix;

matrix reflectionMatrix;

};

cbuffer CamNormBuffer

{

float3 cameraPosition;

float padding1;

float2 normalMapTiling;

float2 padding2;

};

//////////////

// TYPEDEFS //

//////////////

struct VertexInputType

{

float4 position : POSITION;

float2 tex : TEXCOORD0;

};

struct PixelInputType

{

float4 position : SV\_POSITION;

float4 reflectionPosition : TEXCOORD0;

float4 refractionPosition : TEXCOORD1;

float3 viewDirection : TEXCOORD2;

float2 tex1 : TEXCOORD3;

float2 tex2 : TEXCOORD4;

};

////////////////////////////////////////////////////////////////////////////////

// Vertex Shader

////////////////////////////////////////////////////////////////////////////////

PixelInputType WaterVertexShader(VertexInputType input)

{

PixelInputType output;

matrix reflectProjectWorld;

matrix viewProjectWorld;

float4 worldPosition;

Calculate the vertex position as usual.

// Change the position vector to be 4 units for proper matrix calculations.

input.position.w = 1.0f;

// Calculate the position of the vertex against the world, view, and projection matrices.

output.position = mul(input.position, worldMatrix);

output.position = mul(output.position, viewMatrix);

output.position = mul(output.position, projectionMatrix);

Calculate the refraction and reflection view matrices the same as before also.

// Create the reflection projection world matrix.

reflectProjectWorld = mul(reflectionMatrix, projectionMatrix);

reflectProjectWorld = mul(worldMatrix, reflectProjectWorld);

// Calculate the input position against the reflectProjectWorld matrix.

output.reflectionPosition = mul(input.position, reflectProjectWorld);

// Create the view projection world matrix for refraction.

viewProjectWorld = mul(viewMatrix, projectionMatrix);

viewProjectWorld = mul(worldMatrix, viewProjectWorld);

// Calculate the input position against the viewProjectWorld matrix.

output.refractionPosition = mul(input.position, viewProjectWorld);

Calculate the camera's view direction for fresnel and specular calculations.

// Calculate the position of the vertex in the world.

worldPosition = mul(input.position, worldMatrix);

// Determine the viewing direction based on the position of the camera and the position of the vertex in the world.

output.viewDirection = cameraPosition.xyz - worldPosition.xyz;

// Normalize the viewing direction vector.

output.viewDirection = normalize(output.viewDirection);

Calculate two different tiling texture coordinates for the water normal map.

// Create two different texture sample coordinates for tiling the water normal map over the water quad multiple times.

output.tex1 = input.tex / normalMapTiling.x;

output.tex2 = input.tex / normalMapTiling.y;

return output;

}

Water.ps

////////////////////////////////////////////////////////////////////////////////

// Filename: water.ps

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

SamplerState SampleType;

Texture2D refractionTexture : register(t0);

Texture2D reflectionTexture : register(t1);

Texture2D normalTexture : register(t2);

cbuffer WaterBuffer

{

float4 refractionTint;

float3 lightDirection;

float waterTranslation;

float reflectRefractScale;

float specularShininess;

float2 padding;

};

//////////////

// TYPEDEFS //

//////////////

struct PixelInputType

{

float4 position : SV\_POSITION;

float4 reflectionPosition : TEXCOORD0;

float4 refractionPosition : TEXCOORD1;

float3 viewDirection : TEXCOORD2;

float2 tex1 : TEXCOORD3;

float2 tex2 : TEXCOORD4;

};

////////////////////////////////////////////////////////////////////////////////

// Pixel Shader

////////////////////////////////////////////////////////////////////////////////

float4 WaterPixelShader(PixelInputType input) : SV\_TARGET

{

float4 normalMap1;

float4 normalMap2;

float3 normal1;

float3 normal2;

float3 normal;

float2 refractTexCoord;

float2 reflectTexCoord;

float4 reflectionColor;

float4 refractionColor;

float3 heightView;

float r;

float fresnelFactor;

float4 color;

float3 reflection;

float specular;

Translate the two texture coordinates by the water translation amount.

// Move the position the water normal is sampled from to simulate moving water.

input.tex1.y += waterTranslation;

input.tex2.y += waterTranslation;

Sample the water normal map two times using the two different texture sampling coordinates.

// Sample the normal from the normal map texture using the two different tiled and translated coordinates.

normalMap1 = normalTexture.Sample(SampleType, input.tex1);

normalMap2 = normalTexture.Sample(SampleType, input.tex2);

// Expand the range of the normal from (0,1) to (-1,+1).

normal1 = (normalMap1.rgb \* 2.0f) - 1.0f;

normal2 = (normalMap2.rgb \* 2.0f) - 1.0f;

Now combine the two normal map results to get an animated water ripple effect instead of just a single rotated normal map ripple.

// Combine the normals to add the normal maps together.

normal = normalize(normal1 + normal2);

Calculate the sampling coordinates for the refraction and reflection and then sample the textures as we did previously.

// Calculate the projected refraction texture coordinates.

refractTexCoord.x = input.refractionPosition.x / input.refractionPosition.w / 2.0f + 0.5f;

refractTexCoord.y = -input.refractionPosition.y / input.refractionPosition.w / 2.0f + 0.5f;

// Calculate the projected reflection texture coordinates.

reflectTexCoord.x = input.reflectionPosition.x / input.reflectionPosition.w / 2.0f + 0.5f;

reflectTexCoord.y = -input.reflectionPosition.y / input.reflectionPosition.w / 2.0f + 0.5f;

// Re-position the texture coordinate sampling position by the scaled normal map value to simulate the rippling wave effect.

reflectTexCoord = reflectTexCoord + (normal.xy \* reflectRefractScale);

refractTexCoord = refractTexCoord + (normal.xy \* reflectRefractScale);

// Sample the texture pixels from the textures using the updated texture coordinates.

reflectionColor = reflectionTexture.Sample(SampleType, reflectTexCoord);

refractionColor = refractionTexture.Sample(SampleType, refractTexCoord);

Add a water color tint to the refraction.

// Combine the tint with the refraction color.

refractionColor = saturate(refractionColor \* refractionTint);

Create just a height based vector for the fresnel calculation.

// Get a modified viewing direction of the camera that only takes into account height.

heightView.x = input.viewDirection.y;

heightView.y = input.viewDirection.y;

heightView.z = input.viewDirection.y;

Calculate the fresnel factor and then combine the refraction and reflection values based on the fresnel factor.

// Now calculate the fresnel term based solely on height.

r = (1.2f - 1.0f) / (1.2f + 1.0f);

fresnelFactor = max(0.0f, min(1.0f, r + (1.0f - r) \* pow(1.0f - dot(normal, heightView), 2)));

// Combine the reflection and refraction results for the final color using the fresnel factor.

color = lerp(reflectionColor, refractionColor, fresnelFactor);

Finally do a specular light calculation using the water normals and add it to the final color result to get the specular effect on just the water ripples.

// Calculate the reflection vector using the normal and the direction of the light.

reflection = -reflect(normalize(lightDirection), normal);

// Calculate the specular light based on the reflection and the camera position.

specular = dot(normalize(reflection), normalize(input.viewDirection));

// Check to make sure the specular was positive so we aren't adding black spots to the water.

if(specular > 0.0f)

{

// Increase the specular light by the shininess value.

specular = pow(specular, specularShininess);

// Add the specular to the final color.

color = saturate(color + specular);

}

return color;

}

Watershaderclass.h

The WaterShaderClass is the same as it was in the DirectX water tutorial except that it contains variables for the new settings such as specular, texture tiling, and refraction tint.

////////////////////////////////////////////////////////////////////////////////

// Filename: watershaderclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_WATERSHADERCLASS\_H\_

#define \_WATERSHADERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <d3dx11async.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: WaterShaderClass

////////////////////////////////////////////////////////////////////////////////

class WaterShaderClass

{

private:

struct MatrixBufferType

{

D3DXMATRIX world;

D3DXMATRIX view;

D3DXMATRIX projection;

D3DXMATRIX reflection;

};

struct CamNormBufferType

{

D3DXVECTOR3 cameraPosition;

float padding1;

D3DXVECTOR2 normalMapTiling;

D3DXVECTOR2 padding2;

};

struct WaterBufferType

{

D3DXVECTOR4 refractionTint;

D3DXVECTOR3 lightDirection;

float waterTranslation;

float reflectRefractScale;

float specularShininess;

D3DXVECTOR2 padding;

};

public:

WaterShaderClass();

WaterShaderClass(const WaterShaderClass&);

~WaterShaderClass();

bool Initialize(ID3D11Device\*, HWND);

void Shutdown();

bool Render(ID3D11DeviceContext\*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*,

ID3D11ShaderResourceView\*, D3DXVECTOR3, D3DXVECTOR2, float, float, D3DXVECTOR4, D3DXVECTOR3, float);

private:

bool InitializeShader(ID3D11Device\*, HWND, WCHAR\*, WCHAR\*);

void ShutdownShader();

void OutputShaderErrorMessage(ID3D10Blob\*, HWND, WCHAR\*);

bool SetShaderParameters(ID3D11DeviceContext\*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*,

ID3D11ShaderResourceView\*, D3DXVECTOR3, D3DXVECTOR2, float, float, D3DXVECTOR4, D3DXVECTOR3, float);

void RenderShader(ID3D11DeviceContext\*, int);

private:

ID3D11VertexShader\* m\_vertexShader;

ID3D11PixelShader\* m\_pixelShader;

ID3D11InputLayout\* m\_layout;

ID3D11SamplerState\* m\_sampleState;

ID3D11Buffer\* m\_matrixBuffer;

ID3D11Buffer\* m\_camNormBuffer;

ID3D11Buffer\* m\_waterBuffer;

};

#endif

Watershaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: watershaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "watershaderclass.h"

WaterShaderClass::WaterShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

m\_camNormBuffer = 0;

m\_waterBuffer = 0;

}

WaterShaderClass::WaterShaderClass(const WaterShaderClass& other)

{

}

WaterShaderClass::~WaterShaderClass()

{

}

bool WaterShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

// Initialize the vertex and pixel shaders.

result = InitializeShader(device, hwnd, L"../Engine/water.vs", L"../Engine/water.ps");

if(!result)

{

return false;

}

return true;

}

void WaterShaderClass::Shutdown()

{

// Shutdown the vertex and pixel shaders as well as the related objects.

ShutdownShader();

return;

}

bool WaterShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, D3DXMATRIX reflectionMatrix, ID3D11ShaderResourceView\* refractionTexture,

ID3D11ShaderResourceView\* reflectionTexture, ID3D11ShaderResourceView\* normalTexture, D3DXVECTOR3 cameraPosition,

D3DXVECTOR2 normalMapTiling, float waterTranslation, float reflectRefractScale, D3DXVECTOR4 refractionTint,

D3DXVECTOR3 lightDirection, float specularShininess)

{

bool result;

// Set the shader parameters that it will use for rendering.

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, reflectionMatrix, refractionTexture, reflectionTexture,

normalTexture, cameraPosition, normalMapTiling, waterTranslation, reflectRefractScale, refractionTint, lightDirection,

specularShininess);

if(!result)

{

return false;

}

// Now render the prepared buffers with the shader.

RenderShader(deviceContext, indexCount);

return true;

}

bool WaterShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[2];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC camNormBufferDesc;

D3D11\_BUFFER\_DESC waterBufferDesc;

// Initialize the pointers this function will use to null.

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

// Compile the vertex shader code.

result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "WaterVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&vertexShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

// If there was nothing in the error message then it simply could not find the shader file itself.

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Compile the pixel shader code.

result = D3DX11CompileFromFile(psFilename, NULL, NULL, "WaterPixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&pixelShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

// If there was nothing in the error message then it simply could not find the file itself.

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Create the vertex shader from the buffer.

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

// Create the vertex shader from the buffer.

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;

}

// Create the vertex input layout description.

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

// Get a count of the elements in the layout.

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

// Create the vertex input layout.

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), &m\_layout);

if(FAILED(result))

{

return false;

}

// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

// Create a texture sampler state description.

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

// Create the texture sampler state.

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;

}

// Setup the description of the matrix dynamic constant buffer that is in the vertex shader.

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

// Create the matrix constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;

}

// Setup the description of the camera and normal tiling dynamic constant buffer that is in the vertex shader.

camNormBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

camNormBufferDesc.ByteWidth = sizeof(CamNormBufferType);

camNormBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

camNormBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

camNormBufferDesc.MiscFlags = 0;

camNormBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&camNormBufferDesc, NULL, &m\_camNormBuffer);

if(FAILED(result))

{

return false;

}

// Setup the description of the water dynamic constant buffer that is in the pixel shader.

waterBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

waterBufferDesc.ByteWidth = sizeof(WaterBufferType);

waterBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

waterBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

waterBufferDesc.MiscFlags = 0;

waterBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the pixel shader constant buffer from within this class.

result = device->CreateBuffer(&waterBufferDesc, NULL, &m\_waterBuffer);

if(FAILED(result))

{

return false;

}

return true;

}

void WaterShaderClass::ShutdownShader()

{

// Release the water constant buffer.

if(m\_waterBuffer)

{

m\_waterBuffer->Release();

m\_waterBuffer = 0;

}

// Release the camera and normal tiling constant buffer.

if(m\_camNormBuffer)

{

m\_camNormBuffer->Release();

m\_camNormBuffer = 0;

}

// Release the matrix constant buffer.

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;

}

// Release the sampler state.

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;

}

// Release the layout.

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;

}

// Release the pixel shader.

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;

}

// Release the vertex shader.

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;

}

return;

}

void WaterShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

// Get a pointer to the error message text buffer.

compileErrors = (char\*)(errorMessage->GetBufferPointer());

// Get the length of the message.

bufferSize = errorMessage->GetBufferSize();

// Open a file to write the error message to.

fout.open("shader-error.txt");

// Write out the error message.

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

// Close the file.

fout.close();

// Release the error message.

errorMessage->Release();

errorMessage = 0;

// Pop a message up on the screen to notify the user to check the text file for compile errors.

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;

}

bool WaterShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix,

D3DXMATRIX reflectionMatrix, ID3D11ShaderResourceView\* refractionTexture,

ID3D11ShaderResourceView\* reflectionTexture, ID3D11ShaderResourceView\* normalTexture,

D3DXVECTOR3 cameraPosition, D3DXVECTOR2 normalMapTiling, float waterTranslation, float reflectRefractScale,

D3DXVECTOR4 refractionTint, D3DXVECTOR3 lightDirection, float specularShininess)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

unsigned int bufferNumber;

MatrixBufferType\* dataPtr;

CamNormBufferType\* dataPtr2;

WaterBufferType\* dataPtr3;

// Transpose all the input matrices to prepare them for the shader.

D3DXMatrixTranspose(&worldMatrix, &worldMatrix);

D3DXMatrixTranspose(&viewMatrix, &viewMatrix);

D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

D3DXMatrixTranspose(&reflectionMatrix, &reflectionMatrix);

// Lock the matrix constant buffer so it can be written to.

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr = (MatrixBufferType\*)mappedResource.pData;

// Copy the matrices into the constant buffer.

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

dataPtr->reflection = reflectionMatrix;

// Unlock the matrix constant buffer.

deviceContext->Unmap(m\_matrixBuffer, 0);

// Set the position of the matrix constant buffer in the vertex shader.

bufferNumber = 0;

// Now set the matrix constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

// Lock the camera and normal tiling constant buffer so it can be written to.

result = deviceContext->Map(m\_camNormBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr2 = (CamNormBufferType\*)mappedResource.pData;

// Copy the data into the constant buffer.

dataPtr2->cameraPosition = cameraPosition;

dataPtr2->padding1 = 0.0f;

dataPtr2->normalMapTiling = normalMapTiling;

dataPtr2->padding2 = D3DXVECTOR2(0.0f, 0.0f);

// Unlock the constant buffer.

deviceContext->Unmap(m\_camNormBuffer, 0);

// Set the position of the constant buffer in the vertex shader.

bufferNumber = 1;

// Set the constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_camNormBuffer);

// Set the texture resources in the pixel shader.

deviceContext->PSSetShaderResources(0, 1, &refractionTexture);

deviceContext->PSSetShaderResources(1, 1, &reflectionTexture);

deviceContext->PSSetShaderResources(2, 1, &normalTexture);

// Lock the water constant buffer so it can be written to.

result = deviceContext->Map(m\_waterBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr3 = (WaterBufferType\*)mappedResource.pData;

// Copy the water data into the constant buffer.

dataPtr3->waterTranslation = waterTranslation;

dataPtr3->reflectRefractScale = reflectRefractScale;

dataPtr3->refractionTint = refractionTint;

dataPtr3->lightDirection = lightDirection;

dataPtr3->specularShininess = specularShininess;

dataPtr3->padding = D3DXVECTOR2(0.0f, 0.0f);

// Unlock the constant buffer.

deviceContext->Unmap(m\_waterBuffer, 0);

// Set the position of the water constant buffer in the pixel shader.

bufferNumber = 0;

// Finally set the water constant buffer in the pixel shader with the updated values.

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_waterBuffer);

return true;

}

void WaterShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

// Set the vertex input layout.

deviceContext->IASetInputLayout(m\_layout);

// Set the vertex and pixel shaders that will be used to render this triangle.

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

// Set the sampler state in the pixel shader.

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

// Render the triangles.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}

Applicationclass.h

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_APPLICATIONCLASS\_H\_

#define \_APPLICATIONCLASS\_H\_

/////////////

// GLOBALS //

/////////////

const bool FULL\_SCREEN = true;

const bool VSYNC\_ENABLED = true;

const float SCREEN\_DEPTH = 1000.0f;

const float SCREEN\_NEAR = 0.1f;

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "inputclass.h"

#include "d3dclass.h"

#include "timerclass.h"

#include "positionclass.h"

#include "cameraclass.h"

#include "lightclass.h"

#include "terrainclass.h"

#include "terrainshaderclass.h"

#include "skydomeclass.h"

#include "skydomeshaderclass.h"

#include "skyplaneclass.h"

#include "skyplaneshaderclass.h"

#include "fpsclass.h"

#include "cpuclass.h"

#include "fontshaderclass.h"

#include "textclass.h"

Add headers for the render to texture, reflection shader, water, and water shader.

#include "rendertextureclass.h"

#include "reflectionshaderclass.h"

#include "waterclass.h"

#include "watershaderclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: ApplicationClass

////////////////////////////////////////////////////////////////////////////////

class ApplicationClass

{

public:

ApplicationClass();

ApplicationClass(const ApplicationClass&);

~ApplicationClass();

bool Initialize(HINSTANCE, HWND, int, int);

void Shutdown();

bool Frame();

private:

bool HandleMovementInput(float);

void RenderRefractionToTexture();

void RenderReflectionToTexture();

bool Render();

private:

InputClass\* m\_Input;

D3DClass\* m\_Direct3D;

TimerClass\* m\_Timer;

PositionClass\* m\_Position;

CameraClass\* m\_Camera;

LightClass\* m\_Light;

TerrainClass\* m\_Terrain;

TerrainShaderClass\* m\_TerrainShader;

SkyDomeClass\* m\_SkyDome;

SkyDomeShaderClass\* m\_SkyDomeShader;

SkyPlaneClass \*m\_SkyPlane;

SkyPlaneShaderClass\* m\_SkyPlaneShader;

FpsClass\* m\_Fps;

CpuClass\* m\_Cpu;

FontShaderClass\* m\_FontShader;

TextClass\* m\_Text;

There are new objects for the water related rendering.

RenderTextureClass \*m\_RefractionTexture, \*m\_ReflectionTexture;

ReflectionShaderClass\* m\_ReflectionShader;

WaterClass\* m\_Water;

WaterShaderClass\* m\_WaterShader;

};

#endif

Applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "applicationclass.h"

ApplicationClass::ApplicationClass()

{

m\_Input = 0;

m\_Direct3D = 0;

m\_Timer = 0;

m\_Position = 0;

m\_Camera = 0;

m\_Light = 0;

m\_Terrain = 0;

m\_TerrainShader = 0;

m\_SkyDome = 0;

m\_SkyDomeShader = 0;

m\_SkyPlane = 0;

m\_SkyPlaneShader = 0;

m\_Fps = 0;

m\_Cpu = 0;

m\_FontShader = 0;

m\_Text = 0;

m\_RefractionTexture = 0;

m\_ReflectionTexture = 0;

m\_ReflectionShader = 0;

m\_Water = 0;

m\_WaterShader = 0;

}

ApplicationClass::ApplicationClass(const ApplicationClass& other)

{

}

ApplicationClass::~ApplicationClass()

{

}

bool ApplicationClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)

{

bool result;

D3DXMATRIX baseViewMatrix;

char videoCard[128];

int videoMemory;

// Create the input object. The input object will be used to handle reading the keyboard and mouse input from the user.

m\_Input = new InputClass;

if(!m\_Input)

{

return false;

}

// Initialize the input object.

result = m\_Input->Initialize(hinstance, hwnd, screenWidth, screenHeight);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the input object.", L"Error", MB\_OK);

return false;

}

// Create the Direct3D object.

m\_Direct3D = new D3DClass;

if(!m\_Direct3D)

{

return false;

}

// Initialize the Direct3D object.

result = m\_Direct3D->Initialize(screenWidth, screenHeight, VSYNC\_ENABLED, hwnd, FULL\_SCREEN, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize DirectX 11.", L"Error", MB\_OK);

return false;

}

// Retrieve the video card information.

m\_Direct3D->GetVideoCardInfo(videoCard, videoMemory);

// Create the timer object.

m\_Timer = new TimerClass;

if(!m\_Timer)

{

return false;

}

// Initialize the timer object.

result = m\_Timer->Initialize();

if(!result)

{

MessageBox(hwnd, L"Could not initialize the timer object.", L"Error", MB\_OK);

return false;

}

// Create the position object.

m\_Position = new PositionClass;

if(!m\_Position)

{

return false;

}

// Set the initial position and rotation of the viewer.

m\_Position->SetPosition(280.379f, 24.5225f, 367.018f);

m\_Position->SetRotation(19.6834f, 222.013f, 0.0f);

// Create the camera object.

m\_Camera = new CameraClass;

if(!m\_Camera)

{

return false;

}

// Initialize a base view matrix with the camera for 2D user interface rendering.

m\_Camera->SetPosition(0.0f, 0.0f, -10.0f);

m\_Camera->GenerateBaseViewMatrix();

m\_Camera->GetBaseViewMatrix(baseViewMatrix);

// Create the light object.

m\_Light = new LightClass;

if(!m\_Light)

{

return false;

}

// Initialize the light object.

m\_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);

m\_Light->SetDirection(0.5f, -0.75f, 0.25f);

// Create the terrain object.

m\_Terrain = new TerrainClass;

if(!m\_Terrain)

{

return false;

}

// Initialize the terrain object.

result = m\_Terrain->Initialize(m\_Direct3D->GetDevice(), "../Engine/data/hm.bmp", "../Engine/data/cm.bmp", 20.0f, L"../Engine/data/dirt.dds",

L"../Engine/data/normal.dds");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain object.", L"Error", MB\_OK);

return false;

}

// Create the terrain shader object.

m\_TerrainShader = new TerrainShaderClass;

if(!m\_TerrainShader)

{

return false;

}

// Initialize the terrain shader object.

result = m\_TerrainShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain shader object.", L"Error", MB\_OK);

return false;

}

// Create the sky dome object.

m\_SkyDome = new SkyDomeClass;

if(!m\_SkyDome)

{

return false;

}

// Initialize the sky dome object.

result = m\_SkyDome->Initialize(m\_Direct3D->GetDevice());

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome object.", L"Error", MB\_OK);

return false;

}

// Create the sky dome shader object.

m\_SkyDomeShader = new SkyDomeShaderClass;

if(!m\_SkyDomeShader)

{

return false;

}

// Initialize the sky dome shader object.

result = m\_SkyDomeShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky dome shader object.", L"Error", MB\_OK);

return false;

}

// Create the sky plane object.

m\_SkyPlane = new SkyPlaneClass;

if(!m\_SkyPlane)

{

return false;

}

// Initialize the sky plane object.

result = m\_SkyPlane->Initialize(m\_Direct3D->GetDevice(), L"../Engine/data/cloud001.dds", L"../Engine/data/perturb001.dds");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky plane object.", L"Error", MB\_OK);

return false;

}

// Create the sky plane shader object.

m\_SkyPlaneShader = new SkyPlaneShaderClass;

if(!m\_SkyPlaneShader)

{

return false;

}

// Initialize the sky plane shader object.

result = m\_SkyPlaneShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the sky plane shader object.", L"Error", MB\_OK);

return false;

}

// Create the fps object.

m\_Fps = new FpsClass;

if(!m\_Fps)

{

return false;

}

// Initialize the fps object.

m\_Fps->Initialize();

// Create the cpu object.

m\_Cpu = new CpuClass;

if(!m\_Cpu)

{

return false;

}

// Initialize the cpu object.

m\_Cpu->Initialize();

// Create the font shader object.

m\_FontShader = new FontShaderClass;

if(!m\_FontShader)

{

return false;

}

// Initialize the font shader object.

result = m\_FontShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the font shader object.", L"Error", MB\_OK);

return false;

}

// Create the text object.

m\_Text = new TextClass;

if(!m\_Text)

{

return false;

}

// Initialize the text object.

result = m\_Text->Initialize(m\_Direct3D->GetDevice(), m\_Direct3D->GetDeviceContext(), hwnd, screenWidth, screenHeight, baseViewMatrix);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the text object.", L"Error", MB\_OK);

return false;

}

// Set the video card information in the text object.

result = m\_Text->SetVideoCardInfo(videoCard, videoMemory, m\_Direct3D->GetDeviceContext());

if(!result)

{

MessageBox(hwnd, L"Could not set video card info in the text object.", L"Error", MB\_OK);

return false;

}

Setup render to textures for the refraction and reflection of the scene.

// Create the refraction render to texture object.

m\_RefractionTexture = new RenderTextureClass;

if(!m\_RefractionTexture)

{

return false;

}

// Initialize the refraction render to texture object.

result = m\_RefractionTexture->Initialize(m\_Direct3D->GetDevice(), screenWidth, screenHeight, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the refraction render to texture object.", L"Error", MB\_OK);

return false;

}

// Create the reflection render to texture object.

m\_ReflectionTexture = new RenderTextureClass;

if(!m\_ReflectionTexture)

{

return false;

}

// Initialize the reflection render to texture object.

result = m\_ReflectionTexture->Initialize(m\_Direct3D->GetDevice(), screenWidth, screenHeight, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the reflection render to texture object.", L"Error", MB\_OK);

return false;

}

Create the reflection shader for rendering the refraction and the reflection.

// Create the reflection shader object.

m\_ReflectionShader = new ReflectionShaderClass;

if(!m\_ReflectionShader)

{

return false;

}

// Initialize the reflection shader object.

result = m\_ReflectionShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the reflection shader object.", L"Error", MB\_OK);

return false;

}

Setup the WaterClass and WaterShaderClass objects.

// Create the water object.

m\_Water = new WaterClass;

if(!m\_Water)

{

return false;

}

// Initialize the water object.

result = m\_Water->Initialize(m\_Direct3D->GetDevice(), L"../Engine/data/waternormal.dds", 3.75f, 110.0f);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the water object.", L"Error", MB\_OK);

return false;

}

// Create the water shader object.

m\_WaterShader = new WaterShaderClass;

if(!m\_WaterShader)

{

return false;

}

// Initialize the water shader object.

result = m\_WaterShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the water shader object.", L"Error", MB\_OK);

return false;

}

return true;

}

void ApplicationClass::Shutdown()

{

// Release the water shader object.

if(m\_WaterShader)

{

m\_WaterShader->Shutdown();

delete m\_WaterShader;

m\_WaterShader = 0;

}

// Release the water object.

if(m\_Water)

{

m\_Water->Shutdown();

delete m\_Water;

m\_Water = 0;

}

// Release the reflection shader object.

if(m\_ReflectionShader)

{

m\_ReflectionShader->Shutdown();

delete m\_ReflectionShader;

m\_ReflectionShader = 0;

}

// Release the reflection render to texture object.

if(m\_ReflectionTexture)

{

m\_ReflectionTexture->Shutdown();

delete m\_ReflectionTexture;

m\_ReflectionTexture = 0;

}

// Release the refraction render to texture object.

if(m\_RefractionTexture)

{

m\_RefractionTexture->Shutdown();

delete m\_RefractionTexture;

m\_RefractionTexture = 0;

}

// Release the text object.

if(m\_Text)

{

m\_Text->Shutdown();

delete m\_Text;

m\_Text = 0;

}

// Release the font shader object.

if(m\_FontShader)

{

m\_FontShader->Shutdown();

delete m\_FontShader;

m\_FontShader = 0;

}

// Release the cpu object.

if(m\_Cpu)

{

m\_Cpu->Shutdown();

delete m\_Cpu;

m\_Cpu = 0;

}

// Release the fps object.

if(m\_Fps)

{

delete m\_Fps;

m\_Fps = 0;

}

// Release the sky plane shader object.

if(m\_SkyPlaneShader)

{

m\_SkyPlaneShader->Shutdown();

delete m\_SkyPlaneShader;

m\_SkyPlaneShader = 0;

}

// Release the sky plane object.

if(m\_SkyPlane)

{

m\_SkyPlane->Shutdown();

delete m\_SkyPlane;

m\_SkyPlane = 0;

}

// Release the sky dome shader object.

if(m\_SkyDomeShader)

{

m\_SkyDomeShader->Shutdown();

delete m\_SkyDomeShader;

m\_SkyDomeShader = 0;

}

// Release the sky dome object.

if(m\_SkyDome)

{

m\_SkyDome->Shutdown();

delete m\_SkyDome;

m\_SkyDome = 0;

}

// Release the terrain shader object.

if(m\_TerrainShader)

{

m\_TerrainShader->Shutdown();

delete m\_TerrainShader;

m\_TerrainShader = 0;

}

// Release the terrain object.

if(m\_Terrain)

{

m\_Terrain->Shutdown();

delete m\_Terrain;

m\_Terrain = 0;

}

// Release the light object.

if(m\_Light)

{

delete m\_Light;

m\_Light = 0;

}

// Release the camera object.

if(m\_Camera)

{

delete m\_Camera;

m\_Camera = 0;

}

// Release the position object.

if(m\_Position)

{

delete m\_Position;

m\_Position = 0;

}

// Release the timer object.

if(m\_Timer)

{

delete m\_Timer;

m\_Timer = 0;

}

// Release the Direct3D object.

if(m\_Direct3D)

{

m\_Direct3D->Shutdown();

delete m\_Direct3D;

m\_Direct3D = 0;

}

// Release the input object.

if(m\_Input)

{

m\_Input->Shutdown();

delete m\_Input;

m\_Input = 0;

}

return;

}

bool ApplicationClass::Frame()

{

bool result;

// Update the system stats.

m\_Timer->Frame();

m\_Fps->Frame();

m\_Cpu->Frame();

// Read the user input.

result = m\_Input->Frame();

if(!result)

{

return false;

}

// Check if the user pressed escape and wants to exit the application.

if(m\_Input->IsEscapePressed() == true)

{

return false;

}

// Do the frame input processing.

result = HandleMovementInput(m\_Timer->GetTime());

if(!result)

{

return false;

}

// Update the FPS value in the text object.

result = m\_Text->SetFps(m\_Fps->GetFps(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the CPU usage value in the text object.

result = m\_Text->SetCpu(m\_Cpu->GetCpuPercentage(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

The WaterClass object requires frame processing to translate the water normal map sampling.

// Do the water frame processing.

m\_Water->Frame();

// Do the sky plane frame processing.

m\_SkyPlane->Frame();

There are three render steps. First render the refraction of the scene to a texture. Next render the reflection of the scene to a texture. And then finally render the entire scene using the refraction and reflection textures.

// Render the refraction of the scene to a texture.

RenderRefractionToTexture();

// Render the reflection of the scene to a texture.

RenderReflectionToTexture();

// Render the graphics.

result = Render();

if(!result)

{

return false;

}

return result;

}

bool ApplicationClass::HandleMovementInput(float frameTime)

{

bool keyDown, result;

float posX, posY, posZ, rotX, rotY, rotZ;

// Set the frame time for calculating the updated position.

m\_Position->SetFrameTime(frameTime);

// Handle the input.

keyDown = m\_Input->IsLeftPressed();

m\_Position->TurnLeft(keyDown);

keyDown = m\_Input->IsRightPressed();

m\_Position->TurnRight(keyDown);

keyDown = m\_Input->IsUpPressed();

m\_Position->MoveForward(keyDown);

keyDown = m\_Input->IsDownPressed();

m\_Position->MoveBackward(keyDown);

keyDown = m\_Input->IsAPressed();

m\_Position->MoveUpward(keyDown);

keyDown = m\_Input->IsZPressed();

m\_Position->MoveDownward(keyDown);

keyDown = m\_Input->IsPgUpPressed();

m\_Position->LookUpward(keyDown);

keyDown = m\_Input->IsPgDownPressed();

m\_Position->LookDownward(keyDown);

// Get the view point position/rotation.

m\_Position->GetPosition(posX, posY, posZ);

m\_Position->GetRotation(rotX, rotY, rotZ);

// Set the position of the camera.

m\_Camera->SetPosition(posX, posY, posZ);

m\_Camera->SetRotation(rotX, rotY, rotZ);

// Update the position values in the text object.

result = m\_Text->SetCameraPosition(posX, posY, posZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the rotation values in the text object.

result = m\_Text->SetCameraRotation(rotX, rotY, rotZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

return true;

}

Here is where we render the refraction of the scene to a render to texture object. Note we only need to render the terrain as nothing else is below the water other than terrain. We use the clipping plane and then render a refraction of everything under the water using the reflection shader. Note also that we have to offset the sampling of the refraction otherwise we get black spots along the sides of the refraction from sampling outside of the texture.

void ApplicationClass::RenderRefractionToTexture()

{

D3DXVECTOR4 clipPlane;

D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix;

// Setup a clipping plane based on the height of the water to clip everything above it to create a refraction.

clipPlane = D3DXVECTOR4(0.0f, -1.0f, 0.0f, m\_Water->GetWaterHeight() + 0.1f);

// Set the render target to be the refraction render to texture.

m\_RefractionTexture->SetRenderTarget(m\_Direct3D->GetDeviceContext());

// Clear the refraction render to texture.

m\_RefractionTexture->ClearRenderTarget(m\_Direct3D->GetDeviceContext(), 0.0f, 0.0f, 0.0f, 1.0f);

// Generate the view matrix based on the camera's position.

m\_Camera->Render();

// Get the matrices from the camera and d3d objects.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

// Render the terrain using the reflection shader and the refraction clip plane to produce the refraction effect.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

m\_ReflectionShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_Terrain->GetColorTexture(), m\_Terrain->GetNormalTexture(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(), 2.0f,

clipPlane);

// Reset the render target back to the original back buffer and not the render to texture anymore.

m\_Direct3D->SetBackBufferRenderTarget();

// Reset the viewport back to the original.

m\_Direct3D->ResetViewport();

return;

}

Here is where we render the reflection of the scene to a texture. We render everything above the water to a texture.

void ApplicationClass::RenderReflectionToTexture()

{

D3DXVECTOR4 clipPlane;

D3DXMATRIX reflectionViewMatrix, worldMatrix, projectionMatrix;

D3DXVECTOR3 cameraPosition;

// Setup a clipping plane based on the height of the water to clip everything below it.

clipPlane = D3DXVECTOR4(0.0f, 1.0f, 0.0f, -m\_Water->GetWaterHeight());

// Set the render target to be the reflection render to texture.

m\_ReflectionTexture->SetRenderTarget(m\_Direct3D->GetDeviceContext());

// Clear the reflection render to texture.

m\_ReflectionTexture->ClearRenderTarget(m\_Direct3D->GetDeviceContext(), 0.0f, 0.0f, 0.0f, 1.0f);

// Use the camera to render the reflection and create a reflection view matrix.

m\_Camera->RenderReflection(m\_Water->GetWaterHeight());

// Get the camera reflection view matrix instead of the normal view matrix.

m\_Camera->GetReflectionViewMatrix(reflectionViewMatrix);

// Get the world and projection matrices from the d3d object.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

// Get the position of the camera.

cameraPosition = m\_Camera->GetPosition();

// Invert the Y coordinate of the camera around the water plane height for the reflected camera position.

cameraPosition.y = -cameraPosition.y + (m\_Water->GetWaterHeight() \* 2.0f);

// Translate the sky dome and sky plane to be centered around the reflected camera position.

D3DXMatrixTranslation(&worldMatrix, cameraPosition.x, cameraPosition.y, cameraPosition.z);

// Turn off back face culling and the Z buffer.

m\_Direct3D->TurnOffCulling();

m\_Direct3D->TurnZBufferOff();

// Render the sky dome using the reflection view matrix.

m\_SkyDome->Render(m\_Direct3D->GetDeviceContext());

m\_SkyDomeShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyDome->GetIndexCount(), worldMatrix, reflectionViewMatrix, projectionMatrix,

m\_SkyDome->GetApexColor(), m\_SkyDome->GetCenterColor());

// Enable back face culling.

m\_Direct3D->TurnOnCulling();

// Enable additive blending so the clouds blend with the sky dome color.

m\_Direct3D->EnableSecondBlendState();

// Render the sky plane using the sky plane shader.

m\_SkyPlane->Render(m\_Direct3D->GetDeviceContext());

m\_SkyPlaneShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyPlane->GetIndexCount(), worldMatrix, reflectionViewMatrix, projectionMatrix,

m\_SkyPlane->GetCloudTexture(), m\_SkyPlane->GetPerturbTexture(), m\_SkyPlane->GetTranslation(), m\_SkyPlane->GetScale(),

m\_SkyPlane->GetBrightness());

// Turn off blending and enable the Z buffer again.

m\_Direct3D->TurnOffAlphaBlending();

m\_Direct3D->TurnZBufferOn();

// Reset the world matrix.

m\_Direct3D->GetWorldMatrix(worldMatrix);

// Render the terrain using the reflection view matrix and reflection clip plane.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

m\_ReflectionShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, reflectionViewMatrix, projectionMatrix,

m\_Terrain->GetColorTexture(), m\_Terrain->GetNormalTexture(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(), 2.0f,

clipPlane);

// Reset the render target back to the original back buffer and not the render to texture anymore.

m\_Direct3D->SetBackBufferRenderTarget();

// Reset the viewport back to the original.

m\_Direct3D->ResetViewport();

return;

}

Now we render the scene to the back buffer. Everything is rendered as normal except for the water which uses the refraction and reflection render to texture objects as well as the other water related shader parameters.

bool ApplicationClass::Render()

{

D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix, baseViewMatrix, reflectionViewMatrix;

bool result;

D3DXVECTOR3 cameraPosition;

// Clear the scene.

m\_Direct3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

// Generate the view matrix based on the camera's position.

m\_Camera->Render();

// Generate the reflection matrix based on the camera's position and the height of the water.

m\_Camera->RenderReflection(m\_Water->GetWaterHeight());

// Get the world, view, projection, ortho, and base view matrices from the camera and Direct3D objects.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

m\_Direct3D->GetOrthoMatrix(orthoMatrix);

m\_Camera->GetBaseViewMatrix(baseViewMatrix);

m\_Camera->GetReflectionViewMatrix(reflectionViewMatrix);

// Get the position of the camera.

cameraPosition = m\_Camera->GetPosition();

// Translate the sky dome to be centered around the camera position.

D3DXMatrixTranslation(&worldMatrix, cameraPosition.x, cameraPosition.y, cameraPosition.z);

// Turn off back face culling and the Z buffer.

m\_Direct3D->TurnOffCulling();

m\_Direct3D->TurnZBufferOff();

// Render the sky dome using the sky dome shader.

m\_SkyDome->Render(m\_Direct3D->GetDeviceContext());

m\_SkyDomeShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyDome->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyDome->GetApexColor(), m\_SkyDome->GetCenterColor());

// Turn back face culling back on.

m\_Direct3D->TurnOnCulling();

// Enable additive blending so the clouds blend with the sky dome color.

m\_Direct3D->EnableSecondBlendState();

// Render the sky plane using the sky plane shader.

m\_SkyPlane->Render(m\_Direct3D->GetDeviceContext());

m\_SkyPlaneShader->Render(m\_Direct3D->GetDeviceContext(), m\_SkyPlane->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_SkyPlane->GetCloudTexture(), m\_SkyPlane->GetPerturbTexture(), m\_SkyPlane->GetTranslation(), m\_SkyPlane->GetScale(),

m\_SkyPlane->GetBrightness());

// Turn off blending.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on.

m\_Direct3D->TurnZBufferOn();

// Reset the world matrix.

m\_Direct3D->GetWorldMatrix(worldMatrix);

// Render the terrain using the terrain shader.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

result = m\_TerrainShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_Terrain->GetColorTexture(), m\_Terrain->GetNormalTexture(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(),

2.0f);

if(!result)

{

return false;

}

// Translate to the location of the water and render it.

D3DXMatrixTranslation(&worldMatrix, 240.0f, m\_Water->GetWaterHeight(), 250.0f);

m\_Water->Render(m\_Direct3D->GetDeviceContext());

m\_WaterShader->Render(m\_Direct3D->GetDeviceContext(), m\_Water->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix, reflectionViewMatrix,

m\_RefractionTexture->GetShaderResourceView(), m\_ReflectionTexture->GetShaderResourceView(), m\_Water->GetTexture(),

m\_Camera->GetPosition(), m\_Water->GetNormalMapTiling(), m\_Water->GetWaterTranslation(), m\_Water->GetReflectRefractScale(),

m\_Water->GetRefractionTint(), m\_Light->GetDirection(), m\_Water->GetSpecularShininess());

// Reset the world matrix.

m\_Direct3D->GetWorldMatrix(worldMatrix);

// Turn off the Z buffer to begin all 2D rendering.

m\_Direct3D->TurnZBufferOff();

// Turn on the alpha blending before rendering the text.

m\_Direct3D->TurnOnAlphaBlending();

// Render the text user interface elements.

result = m\_Text->Render(m\_Direct3D->GetDeviceContext(), m\_FontShader, worldMatrix, orthoMatrix);

if(!result)

{

return false;

}

// Turn off alpha blending after rendering the text.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on now that all 2D rendering has completed.

m\_Direct3D->TurnZBufferOn();

// Present the rendered scene to the screen.

m\_Direct3D->EndScene();

return true;

}

Summary

We can now render a fairly realistic water effect for small body water that doesn't require any large waves.

To Do Exercises

1. Compile and run the program. Move around using the arrow keys, A, Z, and PgUp and PgDn. Press escape to quit.

2. Modify the shader input values (such as the refraction tint) to see how they change the water effect.

3. Modify the water shader to see each piece of the effect output by itself. For example return only the refraction color, and then return only the reflection color.

4. Optimize the reflection portion using one of the methods described (such as updating the reflection once a second only).

5. Combine this tutorial with the Glass and Ice DirectX tutorial. Render a snowy terrain and change the water to be ice instead.

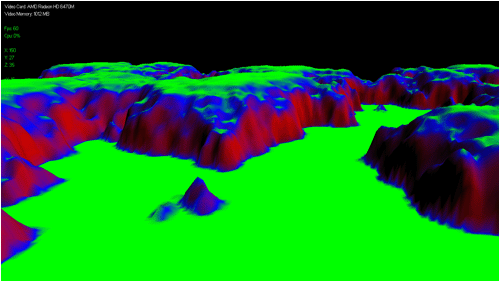
**Tutorial 14: Slope Based Texturing**

http://web.archive.org/web/20140722211314/http:/rastertek.com/pic1001.gif

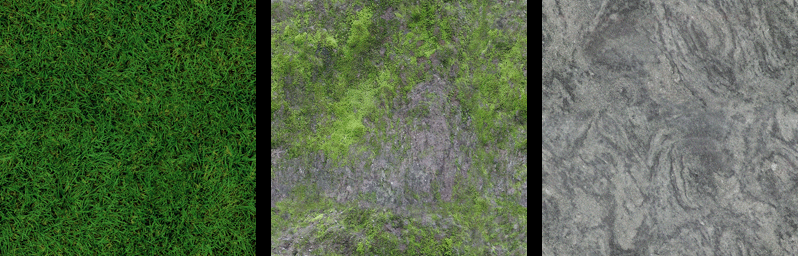
This DirectX 11 terrain tutorial will cover one of the procedural methods for texturing terrain that uses slope as the determining factor. The code in this tutorial is based on the terrain texturing tutorial.

There are a number of applications that use procedural methods to texture terrain. This helps generate large amounts of terrain that look very realistic without the need for an artist. Terragen is one example of such an application that uses procedural texturing. If you are not familiar with Terragen I would definitely recommend taking a look at it. The procedural method that I am going to cover for this tutorial uses slope to determine how to texture any given pixel in the terrain. This is a very fast method that produces excellent results.

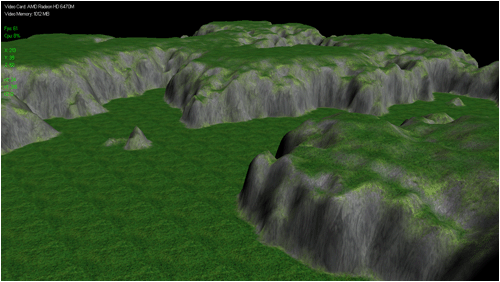
To determine slope for any pixel is simple if you already have the normal vector. You simply subtract one from the Y (height) value of the normal. This will give you a value between 0.0f (completely flat terrain) and 1.0f (terrain pointing straight up at 90 degrees). With the slope for each pixel you can then determine how to texture it. For example lets render some terrain and anything below 0.2f slope is green, anything between 0.2f and 0.7f is a combination of green to blue, and then anything above 0.7 is a combination of blue to red. This would give us the following slope image:



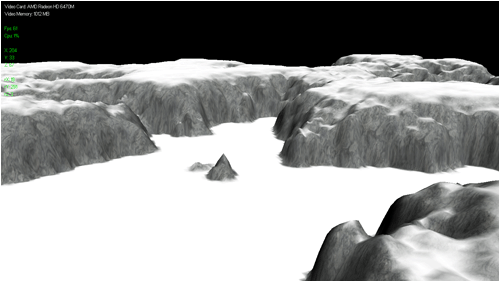
This visualization of slope makes it very clear with red being the extreme slopes, blue being a moderate slope, and green being mostly flat. Now lets take the exact same example further and use the following textures:



Now if we make anything below 0.2f slope the grass texture, and anything between 0.2f and 0.7f a combination of mossy and rock texture, and then anything above 0.7 the rock texture we would get the following appearance:



For another example we could use snow for anything below 0.2 slope and then everything above 0.2 rock and produce the following image with the same terrain:



As you can see slope works great for procedurally rendering textures on terrain. Also it is one of the better systems that correlates with how terrain looks in the real world. For example flat surfaces support growth and retain things such as snow. And high slope areas have less growth and expose more rock and earth.

We will start the code section by looking at the modified terrain shader:

Terrain.vs

The terrain vertex shader has not been modified for this tutorial.

////////////////////////////////////////////////////////////////////////////////

// Filename: terrain.vs

////////////////////////////////////////////////////////////////////////////////

/////////////

// GLOBALS //

/////////////

cbuffer MatrixBuffer

{

matrix worldMatrix;

matrix viewMatrix;

matrix projectionMatrix;

};

//////////////

// TYPEDEFS //

//////////////

struct VertexInputType

{

float4 position : POSITION;

float2 tex : TEXCOORD0;

float3 normal : NORMAL;

};

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

float3 normal : NORMAL;

};

////////////////////////////////////////////////////////////////////////////////

// Vertex Shader

////////////////////////////////////////////////////////////////////////////////

PixelInputType TerrainVertexShader(VertexInputType input)

{

PixelInputType output;

// Change the position vector to be 4 units for proper matrix calculations.

input.position.w = 1.0f;

// Calculate the position of the vertex against the world, view, and projection matrices.

output.position = mul(input.position, worldMatrix);

output.position = mul(output.position, viewMatrix);

output.position = mul(output.position, projectionMatrix);

// Store the texture coordinates for the pixel shader.

output.tex = input.tex;

// Calculate the normal vector against the world matrix only.

output.normal = mul(input.normal, (float3x3)worldMatrix);

// Normalize the normal vector.

output.normal = normalize(output.normal);

return output;

}

Terrain.ps

////////////////////////////////////////////////////////////////////////////////

// Filename: terrain.ps

////////////////////////////////////////////////////////////////////////////////

//////////////

// TEXTURES //

//////////////

There are three textures we will be using for the three different degrees of slope that we want to handle.

Texture2D grassTexture : register(t0);

Texture2D slopeTexture : register(t1);

Texture2D rockTexture : register(t2);

///////////////////

// SAMPLE STATES //

///////////////////

SamplerState SampleType;

//////////////////////

// CONSTANT BUFFERS //

//////////////////////

cbuffer LightBuffer

{

float4 ambientColor;

float4 diffuseColor;

float3 lightDirection;

float padding;

};

//////////////

// TYPEDEFS //

//////////////

struct PixelInputType

{

float4 position : SV\_POSITION;

float2 tex : TEXCOORD0;

float3 normal : NORMAL;

};

////////////////////////////////////////////////////////////////////////////////

// Pixel Shader

////////////////////////////////////////////////////////////////////////////////

float4 TerrainPixelShader(PixelInputType input) : SV\_TARGET

{

float4 grassColor;

float4 slopeColor;

float4 rockColor;

float slope;

float blendAmount;

float4 textureColor;

float3 lightDir;

float lightIntensity;

float4 color;

Start by sampling all three textures.

// Sample the grass color from the texture using the sampler at this texture coordinate location.

grassColor = grassTexture.Sample(SampleType, input.tex);

// Sample the slope color from the texture using the sampler at this texture coordinate location.

slopeColor = slopeTexture.Sample(SampleType, input.tex);

// Sample the rock color from the texture using the sampler at this texture coordinate location.

rockColor = rockTexture.Sample(SampleType, input.tex);

Now determine the slope for this pixel, which is just one subtracted from the Y normal.

// Calculate the slope of this point.

slope = 1.0f - input.normal.y;

Since we have the slope we can now use it in some if statements and determine which texture to use based on the slope of the pixel. To make things look smooth we do a linear interpolation between the textures so the transition between each one isn't a sharp line in the terrain.

// Determine which texture to use based on height.

if(slope < 0.2)

{

blendAmount = slope / 0.2f;

textureColor = lerp(grassColor, slopeColor, blendAmount);

}

if((slope < 0.7) && (slope >= 0.2f))

{

blendAmount = (slope - 0.2f) \* (1.0f / (0.7f - 0.2f));

textureColor = lerp(slopeColor, rockColor, blendAmount);

}

if(slope >= 0.7)

{

textureColor = rockColor;

}

Now do the regular lighting and add the lighting value to the texture value to get the final output color.

// Invert the light direction for calculations.

lightDir = -lightDirection;

// Calculate the amount of light on this pixel.

lightIntensity = saturate(dot(input.normal, lightDir));

// Determine the final diffuse color based on the diffuse color and the amount of light intensity.

color = diffuseColor \* lightIntensity;

// Saturate the final light color.

color = saturate(color);

// Multiply the texture color and the final light color to get the result.

color = color \* textureColor;

return color;

}

Terrainshaderclass.h

The header file has changed a bit to accommodate the extra textures that are set in the shader.

////////////////////////////////////////////////////////////////////////////////

// Filename: terrainshaderclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_TERRAINSHADERCLASS\_H\_

#define \_TERRAINSHADERCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <d3dx11async.h>

#include <fstream>

using namespace std;

////////////////////////////////////////////////////////////////////////////////

// Class name: TerrainShaderClass

////////////////////////////////////////////////////////////////////////////////

class TerrainShaderClass

{

private:

struct MatrixBufferType

{

D3DXMATRIX world;

D3DXMATRIX view;

D3DXMATRIX projection;

};

struct LightBufferType

{

D3DXVECTOR4 ambientColor;

D3DXVECTOR4 diffuseColor;

D3DXVECTOR3 lightDirection;

float padding;

};

public:

TerrainShaderClass();

TerrainShaderClass(const TerrainShaderClass&);

~TerrainShaderClass();

bool Initialize(ID3D11Device\*, HWND);

void Shutdown();

bool Render(ID3D11DeviceContext\*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, D3DXVECTOR4, D3DXVECTOR4, D3DXVECTOR3, ID3D11ShaderResourceView\*,

ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*);

private:

bool InitializeShader(ID3D11Device\*, HWND, WCHAR\*, WCHAR\*);

void ShutdownShader();

void OutputShaderErrorMessage(ID3D10Blob\*, HWND, WCHAR\*);

bool SetShaderParameters(ID3D11DeviceContext\*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX, D3DXVECTOR4, D3DXVECTOR4, D3DXVECTOR3, ID3D11ShaderResourceView\*,

ID3D11ShaderResourceView\*, ID3D11ShaderResourceView\*);

void RenderShader(ID3D11DeviceContext\*, int);

private:

ID3D11VertexShader\* m\_vertexShader;

ID3D11PixelShader\* m\_pixelShader;

ID3D11InputLayout\* m\_layout;

ID3D11SamplerState\* m\_sampleState;

ID3D11Buffer\* m\_matrixBuffer;

ID3D11Buffer\* m\_lightBuffer;

};

#endif

Terrainshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: terrainshaderclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "terrainshaderclass.h"

TerrainShaderClass::TerrainShaderClass()

{

m\_vertexShader = 0;

m\_pixelShader = 0;

m\_layout = 0;

m\_sampleState = 0;

m\_matrixBuffer = 0;

m\_lightBuffer = 0;

}

TerrainShaderClass::TerrainShaderClass(const TerrainShaderClass& other)

{

}

TerrainShaderClass::~TerrainShaderClass()

{

}

bool TerrainShaderClass::Initialize(ID3D11Device\* device, HWND hwnd)

{

bool result;

// Initialize the vertex and pixel shaders.

result = InitializeShader(device, hwnd, L"../Engine/terrain.vs", L"../Engine/terrain.ps");

if(!result)

{

return false;

}

return true;

}

void TerrainShaderClass::Shutdown()

{

// Shutdown the vertex and pixel shaders as well as the related objects.

ShutdownShader();

return;

}

The Render function now takes in the three textures that are used for rendering the terrain based on slope. They are then sent as input into the SetShaderParameters function.

bool TerrainShaderClass::Render(ID3D11DeviceContext\* deviceContext, int indexCount, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, D3DXVECTOR4 ambientColor, D3DXVECTOR4 diffuseColor, D3DXVECTOR3 lightDirection,

ID3D11ShaderResourceView\* grassTexture, ID3D11ShaderResourceView\* slopeTexture, ID3D11ShaderResourceView\* rockTexture)

{

bool result;

// Set the shader parameters that it will use for rendering.

result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, ambientColor, diffuseColor, lightDirection, grassTexture,

slopeTexture, rockTexture);

if(!result)

{

return false;

}

// Now render the prepared buffers with the shader.

RenderShader(deviceContext, indexCount);

return true;

}

bool TerrainShaderClass::InitializeShader(ID3D11Device\* device, HWND hwnd, WCHAR\* vsFilename, WCHAR\* psFilename)

{

HRESULT result;

ID3D10Blob\* errorMessage;

ID3D10Blob\* vertexShaderBuffer;

ID3D10Blob\* pixelShaderBuffer;

D3D11\_INPUT\_ELEMENT\_DESC polygonLayout[3];

unsigned int numElements;

D3D11\_SAMPLER\_DESC samplerDesc;

D3D11\_BUFFER\_DESC matrixBufferDesc;

D3D11\_BUFFER\_DESC lightBufferDesc;

// Initialize the pointers this function will use to null.

errorMessage = 0;

vertexShaderBuffer = 0;

pixelShaderBuffer = 0;

// Compile the vertex shader code.

result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "TerrainVertexShader", "vs\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&vertexShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, vsFilename);

}

// If there was nothing in the error message then it simply could not find the shader file itself.

else

{

MessageBox(hwnd, vsFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Compile the pixel shader code.

result = D3DX11CompileFromFile(psFilename, NULL, NULL, "TerrainPixelShader", "ps\_5\_0", D3D10\_SHADER\_ENABLE\_STRICTNESS, 0, NULL,

&pixelShaderBuffer, &errorMessage, NULL);

if(FAILED(result))

{

// If the shader failed to compile it should have writen something to the error message.

if(errorMessage)

{

OutputShaderErrorMessage(errorMessage, hwnd, psFilename);

}

// If there was nothing in the error message then it simply could not find the file itself.

else

{

MessageBox(hwnd, psFilename, L"Missing Shader File", MB\_OK);

}

return false;

}

// Create the vertex shader from the buffer.

result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(), NULL, &m\_vertexShader);

if(FAILED(result))

{

return false;

}

// Create the pixel shader from the buffer.

result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL, &m\_pixelShader);

if(FAILED(result))

{

return false;

}

// Create the vertex input layout description.

polygonLayout[0].SemanticName = "POSITION";

polygonLayout[0].SemanticIndex = 0;

polygonLayout[0].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[0].InputSlot = 0;

polygonLayout[0].AlignedByteOffset = 0;

polygonLayout[0].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[0].InstanceDataStepRate = 0;

polygonLayout[1].SemanticName = "TEXCOORD";

polygonLayout[1].SemanticIndex = 0;

polygonLayout[1].Format = DXGI\_FORMAT\_R32G32\_FLOAT;

polygonLayout[1].InputSlot = 0;

polygonLayout[1].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[1].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[1].InstanceDataStepRate = 0;

polygonLayout[2].SemanticName = "NORMAL";

polygonLayout[2].SemanticIndex = 0;

polygonLayout[2].Format = DXGI\_FORMAT\_R32G32B32\_FLOAT;

polygonLayout[2].InputSlot = 0;

polygonLayout[2].AlignedByteOffset = D3D11\_APPEND\_ALIGNED\_ELEMENT;

polygonLayout[2].InputSlotClass = D3D11\_INPUT\_PER\_VERTEX\_DATA;

polygonLayout[2].InstanceDataStepRate = 0;

// Get a count of the elements in the layout.

numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

// Create the vertex input layout.

result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(), vertexShaderBuffer->GetBufferSize(),

&m\_layout);

if(FAILED(result))

{

return false;

}

// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.

vertexShaderBuffer->Release();

vertexShaderBuffer = 0;

pixelShaderBuffer->Release();

pixelShaderBuffer = 0;

// Create a texture sampler state description.

samplerDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

samplerDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

samplerDesc.MipLODBias = 0.0f;

samplerDesc.MaxAnisotropy = 1;

samplerDesc.ComparisonFunc = D3D11\_COMPARISON\_ALWAYS;

samplerDesc.BorderColor[0] = 0;

samplerDesc.BorderColor[1] = 0;

samplerDesc.BorderColor[2] = 0;

samplerDesc.BorderColor[3] = 0;

samplerDesc.MinLOD = 0;

samplerDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

// Create the texture sampler state.

result = device->CreateSamplerState(&samplerDesc, &m\_sampleState);

if(FAILED(result))

{

return false;

}

// Setup the description of the dynamic matrix constant buffer that is in the vertex shader.

matrixBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);

matrixBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

matrixBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

matrixBufferDesc.MiscFlags = 0;

matrixBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&matrixBufferDesc, NULL, &m\_matrixBuffer);

if(FAILED(result))

{

return false;

}

// Setup the description of the light dynamic constant buffer that is in the pixel shader.

// Note that ByteWidth always needs to be a multiple of 16 if using D3D11\_BIND\_CONSTANT\_BUFFER or CreateBuffer will fail.

lightBufferDesc.Usage = D3D11\_USAGE\_DYNAMIC;

lightBufferDesc.ByteWidth = sizeof(LightBufferType);

lightBufferDesc.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

lightBufferDesc.CPUAccessFlags = D3D11\_CPU\_ACCESS\_WRITE;

lightBufferDesc.MiscFlags = 0;

lightBufferDesc.StructureByteStride = 0;

// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.

result = device->CreateBuffer(&lightBufferDesc, NULL, &m\_lightBuffer);

if(FAILED(result))

{

return false;

}

return true;

}

void TerrainShaderClass::ShutdownShader()

{

// Release the light constant buffer.

if(m\_lightBuffer)

{

m\_lightBuffer->Release();

m\_lightBuffer = 0;

}

// Release the matrix constant buffer.

if(m\_matrixBuffer)

{

m\_matrixBuffer->Release();

m\_matrixBuffer = 0;

}

// Release the sampler state.

if(m\_sampleState)

{

m\_sampleState->Release();

m\_sampleState = 0;

}

// Release the layout.

if(m\_layout)

{

m\_layout->Release();

m\_layout = 0;

}

// Release the pixel shader.

if(m\_pixelShader)

{

m\_pixelShader->Release();

m\_pixelShader = 0;

}

// Release the vertex shader.

if(m\_vertexShader)

{

m\_vertexShader->Release();

m\_vertexShader = 0;

}

return;

}

void TerrainShaderClass::OutputShaderErrorMessage(ID3D10Blob\* errorMessage, HWND hwnd, WCHAR\* shaderFilename)

{

char\* compileErrors;

unsigned long bufferSize, i;

ofstream fout;

// Get a pointer to the error message text buffer.

compileErrors = (char\*)(errorMessage->GetBufferPointer());

// Get the length of the message.

bufferSize = errorMessage->GetBufferSize();

// Open a file to write the error message to.

fout.open("shader-error.txt");

// Write out the error message.

for(i=0; i<bufferSize; i++)

{

fout << compileErrors[i];

}

// Close the file.

fout.close();

// Release the error message.

errorMessage->Release();

errorMessage = 0;

// Pop a message up on the screen to notify the user to check the text file for compile errors.

MessageBox(hwnd, L"Error compiling shader. Check shader-error.txt for message.", shaderFilename, MB\_OK);

return;

}

SetShaderParameters now takes in the three textures for rendering the terrain.

bool TerrainShaderClass::SetShaderParameters(ID3D11DeviceContext\* deviceContext, D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix,

D3DXMATRIX projectionMatrix, D3DXVECTOR4 ambientColor, D3DXVECTOR4 diffuseColor, D3DXVECTOR3 lightDirection,

ID3D11ShaderResourceView\* grassTexture, ID3D11ShaderResourceView\* slopeTexture,

ID3D11ShaderResourceView\* rockTexture)

{

HRESULT result;

D3D11\_MAPPED\_SUBRESOURCE mappedResource;

unsigned int bufferNumber;

MatrixBufferType\* dataPtr;

LightBufferType\* dataPtr2;

// Transpose the matrices to prepare them for the shader.

D3DXMatrixTranspose(&worldMatrix, &worldMatrix);

D3DXMatrixTranspose(&viewMatrix, &viewMatrix);

D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

// Lock the constant buffer so it can be written to.

result = deviceContext->Map(m\_matrixBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr = (MatrixBufferType\*)mappedResource.pData;

// Copy the matrices into the constant buffer.

dataPtr->world = worldMatrix;

dataPtr->view = viewMatrix;

dataPtr->projection = projectionMatrix;

// Unlock the constant buffer.

deviceContext->Unmap(m\_matrixBuffer, 0);

// Set the position of the constant buffer in the vertex shader.

bufferNumber = 0;

// Now set the constant buffer in the vertex shader with the updated values.

deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m\_matrixBuffer);

// Lock the light constant buffer so it can be written to.

result = deviceContext->Map(m\_lightBuffer, 0, D3D11\_MAP\_WRITE\_DISCARD, 0, &mappedResource);

if(FAILED(result))

{

return false;

}

// Get a pointer to the data in the constant buffer.

dataPtr2 = (LightBufferType\*)mappedResource.pData;

// Copy the lighting variables into the constant buffer.

dataPtr2->ambientColor = ambientColor;

dataPtr2->diffuseColor = diffuseColor;

dataPtr2->lightDirection = lightDirection;

dataPtr2->padding = 0.0f;

// Unlock the constant buffer.

deviceContext->Unmap(m\_lightBuffer, 0);

// Set the position of the light constant buffer in the pixel shader.

bufferNumber = 0;

// Finally set the light constant buffer in the pixel shader with the updated values.

deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m\_lightBuffer);

Set the three textures in the pixel shader here.

// Set shader texture resources in the pixel shader.

deviceContext->PSSetShaderResources(0, 1, &grassTexture);

deviceContext->PSSetShaderResources(1, 1, &slopeTexture);

deviceContext->PSSetShaderResources(2, 1, &rockTexture);

return true;

}

void TerrainShaderClass::RenderShader(ID3D11DeviceContext\* deviceContext, int indexCount)

{

// Set the vertex input layout.

deviceContext->IASetInputLayout(m\_layout);

// Set the vertex and pixel shaders that will be used to render this triangle.

deviceContext->VSSetShader(m\_vertexShader, NULL, 0);

deviceContext->PSSetShader(m\_pixelShader, NULL, 0);

// Set the sampler state in the pixel shader.

deviceContext->PSSetSamplers(0, 1, &m\_sampleState);

// Render the triangle.

deviceContext->DrawIndexed(indexCount, 0, 0);

return;

}

Terrainclass.h

////////////////////////////////////////////////////////////////////////////////

// Filename: terrainclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_TERRAINCLASS\_H\_

#define \_TERRAINCLASS\_H\_

//////////////

// INCLUDES //

//////////////

#include <d3d11.h>

#include <d3dx10math.h>

#include <stdio.h>

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "textureclass.h"

/////////////

// GLOBALS //

/////////////

const int TEXTURE\_REPEAT = 32;

////////////////////////////////////////////////////////////////////////////////

// Class name: TerrainClass

////////////////////////////////////////////////////////////////////////////////

class TerrainClass

{

private:

struct VertexType

{

D3DXVECTOR3 position;

D3DXVECTOR2 texture;

D3DXVECTOR3 normal;

};

struct HeightMapType

{

float x, y, z;

float tu, tv;

float nx, ny, nz;

};

struct VectorType

{

float x, y, z;

};

public:

TerrainClass();

TerrainClass(const TerrainClass&);

~TerrainClass();

bool Initialize(ID3D11Device\*, char\*, WCHAR\*, WCHAR\*, WCHAR\*);

void Shutdown();

void Render(ID3D11DeviceContext\*);

int GetIndexCount();

The TerrainClass now has functions for returning pointers to the three new texture resources that are used to render the terrain.

ID3D11ShaderResourceView\* GetGrassTexture();

ID3D11ShaderResourceView\* GetSlopeTexture();

ID3D11ShaderResourceView\* GetRockTexture();

private:

bool LoadHeightMap(char\*);

void NormalizeHeightMap();

bool CalculateNormals();

void ShutdownHeightMap();

void CalculateTextureCoordinates();

bool LoadTextures(ID3D11Device\*, WCHAR\*, WCHAR\*, WCHAR\*);

void ReleaseTextures();

bool InitializeBuffers(ID3D11Device\*);

void ShutdownBuffers();

void RenderBuffers(ID3D11DeviceContext\*);

private:

int m\_terrainWidth, m\_terrainHeight;

int m\_vertexCount, m\_indexCount;

ID3D11Buffer \*m\_vertexBuffer, \*m\_indexBuffer;

HeightMapType\* m\_heightMap;

We have three new TextureClass objects for the three new textures used for terrain rendering.

TextureClass \*m\_GrassTexture, \*m\_SlopeTexture, \*m\_RockTexture;

};

#endif

Terrainclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: terrainclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "terrainclass.h"

TerrainClass::TerrainClass()

{

m\_vertexBuffer = 0;

m\_indexBuffer = 0;

m\_heightMap = 0;

The three new textures are set to null in the class constructor.

m\_GrassTexture = 0;

m\_SlopeTexture = 0;

m\_RockTexture = 0;

}

TerrainClass::TerrainClass(const TerrainClass& other)

{

}

TerrainClass::~TerrainClass()

{

}

bool TerrainClass::Initialize(ID3D11Device\* device, char\* heightMapFilename, WCHAR\* grassTextureFilename, WCHAR\* slopeTextureFilename,

WCHAR\* rockTextureFilename)

{

bool result;

// Load in the height map for the terrain.

result = LoadHeightMap(heightMapFilename);

if(!result)

{

return false;

}

// Normalize the height of the height map.

NormalizeHeightMap();

// Calculate the normals for the terrain data.

result = CalculateNormals();

if(!result)

{

return false;

}

// Calculate the texture coordinates.

CalculateTextureCoordinates();

The file names of the three new textures are sent into the LoadTextures function.

// Load the textures.

result = LoadTextures(device, grassTextureFilename, slopeTextureFilename, rockTextureFilename);

if(!result)

{

return false;

}

// Initialize the vertex and index buffer that hold the geometry for the terrain.

result = InitializeBuffers(device);

if(!result)

{

return false;

}

return true;

}

void TerrainClass::Shutdown()

{

// Release the textures.

ReleaseTextures();

// Release the vertex and index buffer.

ShutdownBuffers();

// Release the height map data.

ShutdownHeightMap();

return;

}

void TerrainClass::Render(ID3D11DeviceContext\* deviceContext)

{

// Put the vertex and index buffers on the graphics pipeline to prepare them for drawing.

RenderBuffers(deviceContext);

return;

}

int TerrainClass::GetIndexCount()

{

return m\_indexCount;

}

These are the three new functions that return pointers to the texture resources that are used in the pixel shader.

ID3D11ShaderResourceView\* TerrainClass::GetGrassTexture()

{

return m\_GrassTexture->GetTexture();

}

ID3D11ShaderResourceView\* TerrainClass::GetSlopeTexture()

{

return m\_SlopeTexture->GetTexture();

}

ID3D11ShaderResourceView\* TerrainClass::GetRockTexture()

{

return m\_RockTexture->GetTexture();

}

bool TerrainClass::LoadHeightMap(char\* filename)

{

FILE\* filePtr;

int error;

unsigned int count;

BITMAPFILEHEADER bitmapFileHeader;

BITMAPINFOHEADER bitmapInfoHeader;

int imageSize, i, j, k, index;

unsigned char\* bitmapImage;

unsigned char height;

// Open the height map file in binary.

error = fopen\_s(&filePtr, filename, "rb");

if(error != 0)

{

return false;

}

// Read in the file header.

count = fread(&bitmapFileHeader, sizeof(BITMAPFILEHEADER), 1, filePtr);

if(count != 1)

{

return false;

}

// Read in the bitmap info header.

count = fread(&bitmapInfoHeader, sizeof(BITMAPINFOHEADER), 1, filePtr);

if(count != 1)

{

return false;

}

// Save the dimensions of the terrain.

m\_terrainWidth = bitmapInfoHeader.biWidth;

m\_terrainHeight = bitmapInfoHeader.biHeight;

// Calculate the size of the bitmap image data.

imageSize = m\_terrainWidth \* m\_terrainHeight \* 3;

// Allocate memory for the bitmap image data.

bitmapImage = new unsigned char[imageSize];

if(!bitmapImage)

{

return false;

}

// Move to the beginning of the bitmap data.

fseek(filePtr, bitmapFileHeader.bfOffBits, SEEK\_SET);

// Read in the bitmap image data.

count = fread(bitmapImage, 1, imageSize, filePtr);

if(count != imageSize)

{

return false;

}

// Close the file.

error = fclose(filePtr);

if(error != 0)

{

return false;

}

// Create the structure to hold the height map data.

m\_heightMap = new HeightMapType[m\_terrainWidth \* m\_terrainHeight];

if(!m\_heightMap)

{

return false;

}

// Initialize the position in the image data buffer.

k=0;

// Read the image data into the height map.

for(j=0; j<m\_terrainHeight; j++)

{

for(i=0; i<m\_terrainWidth; i++)

{

height = bitmapImage[k];

index = (m\_terrainHeight \* j) + i;

m\_heightMap[index].x = (float)i;

m\_heightMap[index].y = (float)height;

m\_heightMap[index].z = (float)j;

k+=3;

}

}

// Release the bitmap image data.

delete [] bitmapImage;

bitmapImage = 0;

return true;

}

void TerrainClass::NormalizeHeightMap()

{

int i, j;

for(j=0; j<m\_terrainHeight; j++)

{

for(i=0; i<m\_terrainWidth; i++)

{

m\_heightMap[(m\_terrainHeight \* j) + i].y /= 15.0f;

}

}

return;

}

bool TerrainClass::CalculateNormals()

{

int i, j, index1, index2, index3, index, count;

float vertex1[3], vertex2[3], vertex3[3], vector1[3], vector2[3], sum[3], length;

VectorType\* normals;

// Create a temporary array to hold the un-normalized normal vectors.

normals = new VectorType[(m\_terrainHeight-1) \* (m\_terrainWidth-1)];

if(!normals)

{

return false;

}

// Go through all the faces in the mesh and calculate their normals.

for(j=0; j<(m\_terrainHeight-1); j++)

{

for(i=0; i<(m\_terrainWidth-1); i++)

{

index1 = (j \* m\_terrainHeight) + i;

index2 = (j \* m\_terrainHeight) + (i+1);

index3 = ((j+1) \* m\_terrainHeight) + i;

// Get three vertices from the face.

vertex1[0] = m\_heightMap[index1].x;

vertex1[1] = m\_heightMap[index1].y;

vertex1[2] = m\_heightMap[index1].z;

vertex2[0] = m\_heightMap[index2].x;

vertex2[1] = m\_heightMap[index2].y;

vertex2[2] = m\_heightMap[index2].z;

vertex3[0] = m\_heightMap[index3].x;

vertex3[1] = m\_heightMap[index3].y;

vertex3[2] = m\_heightMap[index3].z;

// Calculate the two vectors for this face.

vector1[0] = vertex1[0] - vertex3[0];

vector1[1] = vertex1[1] - vertex3[1];

vector1[2] = vertex1[2] - vertex3[2];

vector2[0] = vertex3[0] - vertex2[0];

vector2[1] = vertex3[1] - vertex2[1];

vector2[2] = vertex3[2] - vertex2[2];

index = (j \* (m\_terrainHeight-1)) + i;

// Calculate the cross product of those two vectors to get the un-normalized value for this face normal.

normals[index].x = (vector1[1] \* vector2[2]) - (vector1[2] \* vector2[1]);

normals[index].y = (vector1[2] \* vector2[0]) - (vector1[0] \* vector2[2]);

normals[index].z = (vector1[0] \* vector2[1]) - (vector1[1] \* vector2[0]);

}

}

// Now go through all the vertices and take an average of each face normal

// that the vertex touches to get the averaged normal for that vertex.

for(j=0; j<m\_terrainHeight; j++)

{

for(i=0; i<m\_terrainWidth; i++)

{

// Initialize the sum.

sum[0] = 0.0f;

sum[1] = 0.0f;

sum[2] = 0.0f;

// Initialize the count.

count = 0;

// Bottom left face.

if(((i-1) >= 0) && ((j-1) >= 0))

{

index = ((j-1) \* (m\_terrainHeight-1)) + (i-1);

sum[0] += normals[index].x;

sum[1] += normals[index].y;

sum[2] += normals[index].z;

count++;

}

// Bottom right face.

if((i < (m\_terrainWidth-1)) && ((j-1) >= 0))

{

index = ((j-1) \* (m\_terrainHeight-1)) + i;

sum[0] += normals[index].x;

sum[1] += normals[index].y;

sum[2] += normals[index].z;

count++;

}

// Upper left face.

if(((i-1) >= 0) && (j < (m\_terrainHeight-1)))

{

index = (j \* (m\_terrainHeight-1)) + (i-1);

sum[0] += normals[index].x;

sum[1] += normals[index].y;

sum[2] += normals[index].z;

count++;

}

// Upper right face.

if((i < (m\_terrainWidth-1)) && (j < (m\_terrainHeight-1)))

{

index = (j \* (m\_terrainHeight-1)) + i;

sum[0] += normals[index].x;

sum[1] += normals[index].y;

sum[2] += normals[index].z;

count++;

}

// Take the average of the faces touching this vertex.

sum[0] = (sum[0] / (float)count);

sum[1] = (sum[1] / (float)count);

sum[2] = (sum[2] / (float)count);

// Calculate the length of this normal.

length = sqrt((sum[0] \* sum[0]) + (sum[1] \* sum[1]) + (sum[2] \* sum[2]));

// Get an index to the vertex location in the height map array.

index = (j \* m\_terrainHeight) + i;

// Normalize the final shared normal for this vertex and store it in the height map array.

m\_heightMap[index].nx = (sum[0] / length);

m\_heightMap[index].ny = (sum[1] / length);

m\_heightMap[index].nz = (sum[2] / length);

}

}

// Release the temporary normals.

delete [] normals;

normals = 0;

return true;

}

void TerrainClass::ShutdownHeightMap()

{

if(m\_heightMap)

{

delete [] m\_heightMap;

m\_heightMap = 0;

}

return;

}

void TerrainClass::CalculateTextureCoordinates()

{

int incrementCount, i, j, tuCount, tvCount;

float incrementValue, tuCoordinate, tvCoordinate;

// Calculate how much to increment the texture coordinates by.

incrementValue = (float)TEXTURE\_REPEAT / (float)m\_terrainWidth;

// Calculate how many times to repeat the texture.

incrementCount = m\_terrainWidth / TEXTURE\_REPEAT;

// Initialize the tu and tv coordinate values.

tuCoordinate = 0.0f;

tvCoordinate = 1.0f;

// Initialize the tu and tv coordinate indexes.

tuCount = 0;

tvCount = 0;

// Loop through the entire height map and calculate the tu and tv texture coordinates for each vertex.

for(j=0; j<m\_terrainHeight; j++)

{

for(i=0; i<m\_terrainWidth; i++)

{

// Store the texture coordinate in the height map.

m\_heightMap[(m\_terrainHeight \* j) + i].tu = tuCoordinate;

m\_heightMap[(m\_terrainHeight \* j) + i].tv = tvCoordinate;

// Increment the tu texture coordinate by the increment value and increment the index by one.

tuCoordinate += incrementValue;

tuCount++;

// Check if at the far right end of the texture and if so then start at the beginning again.

if(tuCount == incrementCount)

{

tuCoordinate = 0.0f;

tuCount = 0;

}

}

// Increment the tv texture coordinate by the increment value and increment the index by one.

tvCoordinate -= incrementValue;

tvCount++;

// Check if at the top of the texture and if so then start at the bottom again.

if(tvCount == incrementCount)

{

tvCoordinate = 1.0f;

tvCount = 0;

}

}

return;

}

The three new textures are loaded here.

bool TerrainClass::LoadTextures(ID3D11Device\* device, WCHAR\* grassTextureFilename, WCHAR\* slopeTextureFilename, WCHAR\* rockTextureFilename)

{

bool result;

// Create the grass texture object.

m\_GrassTexture = new TextureClass;

if(!m\_GrassTexture)

{

return false;

}

// Initialize the grass texture object.

result = m\_GrassTexture->Initialize(device, grassTextureFilename);

if(!result)

{

return false;

}

// Create the slope texture object.

m\_SlopeTexture = new TextureClass;

if(!m\_SlopeTexture)

{

return false;

}

// Initialize the slope texture object.

result = m\_SlopeTexture->Initialize(device, slopeTextureFilename);

if(!result)

{

return false;

}

// Create the rock texture object.

m\_RockTexture = new TextureClass;

if(!m\_RockTexture)

{

return false;

}

// Initialize the rock texture object.

result = m\_RockTexture->Initialize(device, rockTextureFilename);

if(!result)

{

return false;

}

return true;

}

The three new textures are released here in the ReleaseTextures function.

void TerrainClass::ReleaseTextures()

{

// Release the texture objects.

if(m\_GrassTexture)

{

m\_GrassTexture->Shutdown();

delete m\_GrassTexture;

m\_GrassTexture = 0;

}

if(m\_SlopeTexture)

{

m\_SlopeTexture->Shutdown();

delete m\_SlopeTexture;

m\_SlopeTexture = 0;

}

if(m\_RockTexture)

{

m\_RockTexture->Shutdown();

delete m\_RockTexture;

m\_RockTexture = 0;

}

return;

}

bool TerrainClass::InitializeBuffers(ID3D11Device\* device)

{

VertexType\* vertices;

unsigned long\* indices;

int index, i, j;

D3D11\_BUFFER\_DESC vertexBufferDesc, indexBufferDesc;

D3D11\_SUBRESOURCE\_DATA vertexData, indexData;

HRESULT result;

int index1, index2, index3, index4;

float tu, tv;

// Calculate the number of vertices in the terrain mesh.

m\_vertexCount = (m\_terrainWidth - 1) \* (m\_terrainHeight - 1) \* 6;

// Set the index count to the same as the vertex count.

m\_indexCount = m\_vertexCount;

// Create the vertex array.

vertices = new VertexType[m\_vertexCount];

if(!vertices)

{

return false;

}

// Create the index array.

indices = new unsigned long[m\_indexCount];

if(!indices)

{

return false;

}

// Initialize the index to the vertex buffer.

index = 0;

// Load the vertex and index array with the terrain data.

for(j=0; j<(m\_terrainHeight-1); j++)

{

for(i=0; i<(m\_terrainWidth-1); i++)

{

index1 = (m\_terrainHeight \* j) + i; // Bottom left.

index2 = (m\_terrainHeight \* j) + (i+1); // Bottom right.

index3 = (m\_terrainHeight \* (j+1)) + i; // Upper left.

index4 = (m\_terrainHeight \* (j+1)) + (i+1); // Upper right.

// Upper left.

tv = m\_heightMap[index3].tv;

// Modify the texture coordinates to cover the top edge.

if(tv == 1.0f) { tv = 0.0f; }

vertices[index].position = D3DXVECTOR3(m\_heightMap[index3].x, m\_heightMap[index3].y, m\_heightMap[index3].z);

vertices[index].texture = D3DXVECTOR2(m\_heightMap[index3].tu, tv);

vertices[index].normal = D3DXVECTOR3(m\_heightMap[index3].nx, m\_heightMap[index3].ny, m\_heightMap[index3].nz);

indices[index] = index;

index++;

// Upper right.

tu = m\_heightMap[index4].tu;

tv = m\_heightMap[index4].tv;

// Modify the texture coordinates to cover the top and right edge.

if(tu == 0.0f) { tu = 1.0f; }

if(tv == 1.0f) { tv = 0.0f; }

vertices[index].position = D3DXVECTOR3(m\_heightMap[index4].x, m\_heightMap[index4].y, m\_heightMap[index4].z);

vertices[index].texture = D3DXVECTOR2(tu, tv);

vertices[index].normal = D3DXVECTOR3(m\_heightMap[index4].nx, m\_heightMap[index4].ny, m\_heightMap[index4].nz);

indices[index] = index;

index++;

// Bottom left.

vertices[index].position = D3DXVECTOR3(m\_heightMap[index1].x, m\_heightMap[index1].y, m\_heightMap[index1].z);

vertices[index].texture = D3DXVECTOR2(m\_heightMap[index1].tu, m\_heightMap[index1].tv);

vertices[index].normal = D3DXVECTOR3(m\_heightMap[index1].nx, m\_heightMap[index1].ny, m\_heightMap[index1].nz);

indices[index] = index;

index++;

// Bottom left.

vertices[index].position = D3DXVECTOR3(m\_heightMap[index1].x, m\_heightMap[index1].y, m\_heightMap[index1].z);

vertices[index].texture = D3DXVECTOR2(m\_heightMap[index1].tu, m\_heightMap[index1].tv);

vertices[index].normal = D3DXVECTOR3(m\_heightMap[index1].nx, m\_heightMap[index1].ny, m\_heightMap[index1].nz);

indices[index] = index;

index++;

// Upper right.

tu = m\_heightMap[index4].tu;

tv = m\_heightMap[index4].tv;

// Modify the texture coordinates to cover the top and right edge.

if(tu == 0.0f) { tu = 1.0f; }

if(tv == 1.0f) { tv = 0.0f; }

vertices[index].position = D3DXVECTOR3(m\_heightMap[index4].x, m\_heightMap[index4].y, m\_heightMap[index4].z);

vertices[index].texture = D3DXVECTOR2(tu, tv);

vertices[index].normal = D3DXVECTOR3(m\_heightMap[index4].nx, m\_heightMap[index4].ny, m\_heightMap[index4].nz);

indices[index] = index;

index++;

// Bottom right.

tu = m\_heightMap[index2].tu;

// Modify the texture coordinates to cover the right edge.

if(tu == 0.0f) { tu = 1.0f; }

vertices[index].position = D3DXVECTOR3(m\_heightMap[index2].x, m\_heightMap[index2].y, m\_heightMap[index2].z);

vertices[index].texture = D3DXVECTOR2(tu, m\_heightMap[index2].tv);

vertices[index].normal = D3DXVECTOR3(m\_heightMap[index2].nx, m\_heightMap[index2].ny, m\_heightMap[index2].nz);

indices[index] = index;

index++;

}

}

// Set up the description of the static vertex buffer.

vertexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

vertexBufferDesc.ByteWidth = sizeof(VertexType) \* m\_vertexCount;

vertexBufferDesc.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

vertexBufferDesc.CPUAccessFlags = 0;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the vertex data.

vertexData.pSysMem = vertices;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

// Now create the vertex buffer.

result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m\_vertexBuffer);

if(FAILED(result))

{

return false;

}

// Set up the description of the static index buffer.

indexBufferDesc.Usage = D3D11\_USAGE\_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) \* m\_indexCount;

indexBufferDesc.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the index data.

indexData.pSysMem = indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

// Create the index buffer.

result = device->CreateBuffer(&indexBufferDesc, &indexData, &m\_indexBuffer);

if(FAILED(result))

{

return false;

}

// Release the arrays now that the buffers have been created and loaded.

delete [] vertices;

vertices = 0;

delete [] indices;

indices = 0;

return true;

}

void TerrainClass::ShutdownBuffers()

{

// Release the index buffer.

if(m\_indexBuffer)

{

m\_indexBuffer->Release();

m\_indexBuffer = 0;

}

// Release the vertex buffer.

if(m\_vertexBuffer)

{

m\_vertexBuffer->Release();

m\_vertexBuffer = 0;

}

return;

}

void TerrainClass::RenderBuffers(ID3D11DeviceContext\* deviceContext)

{

unsigned int stride;

unsigned int offset;

// Set vertex buffer stride and offset.

stride = sizeof(VertexType);

offset = 0;

// Set the vertex buffer to active in the input assembler so it can be rendered.

deviceContext->IASetVertexBuffers(0, 1, &m\_vertexBuffer, &stride, &offset);

// Set the index buffer to active in the input assembler so it can be rendered.

deviceContext->IASetIndexBuffer(m\_indexBuffer, DXGI\_FORMAT\_R32\_UINT, 0);

// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.

deviceContext->IASetPrimitiveTopology(D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST);

return;

}

Applicationclass.h

The ApplicationClass header hasn't changed for this tutorial.

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.h

////////////////////////////////////////////////////////////////////////////////

#ifndef \_APPLICATIONCLASS\_H\_

#define \_APPLICATIONCLASS\_H\_

/////////////

// GLOBALS //

/////////////

const bool FULL\_SCREEN = true;

const bool VSYNC\_ENABLED = true;

const float SCREEN\_DEPTH = 1000.0f;

const float SCREEN\_NEAR = 0.1f;

///////////////////////

// MY CLASS INCLUDES //

///////////////////////

#include "inputclass.h"

#include "d3dclass.h"

#include "cameraclass.h"

#include "terrainclass.h"

#include "timerclass.h"

#include "positionclass.h"

#include "fpsclass.h"

#include "cpuclass.h"

#include "fontshaderclass.h"

#include "textclass.h"

#include "terrainshaderclass.h"

#include "lightclass.h"

////////////////////////////////////////////////////////////////////////////////

// Class name: ApplicationClass

////////////////////////////////////////////////////////////////////////////////

class ApplicationClass

{

public:

ApplicationClass();

ApplicationClass(const ApplicationClass&);

~ApplicationClass();

bool Initialize(HINSTANCE, HWND, int, int);

void Shutdown();

bool Frame();

private:

bool HandleInput(float);

bool RenderGraphics();

private:

InputClass\* m\_Input;

D3DClass\* m\_Direct3D;

CameraClass\* m\_Camera;

TerrainClass\* m\_Terrain;

TimerClass\* m\_Timer;

PositionClass\* m\_Position;

FpsClass\* m\_Fps;

CpuClass\* m\_Cpu;

FontShaderClass\* m\_FontShader;

TextClass\* m\_Text;

TerrainShaderClass\* m\_TerrainShader;

LightClass\* m\_Light;

};

#endif

Applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

// Filename: applicationclass.cpp

////////////////////////////////////////////////////////////////////////////////

#include "applicationclass.h"

ApplicationClass::ApplicationClass()

{

m\_Input = 0;

m\_Direct3D = 0;

m\_Camera = 0;

m\_Terrain = 0;

m\_Timer = 0;

m\_Position = 0;

m\_Fps = 0;

m\_Cpu = 0;

m\_FontShader = 0;

m\_Text = 0;

m\_TerrainShader = 0;

m\_Light = 0;

}

ApplicationClass::ApplicationClass(const ApplicationClass& other)

{

}

ApplicationClass::~ApplicationClass()

{

}

bool ApplicationClass::Initialize(HINSTANCE hinstance, HWND hwnd, int screenWidth, int screenHeight)

{

bool result;

float cameraX, cameraY, cameraZ;

D3DXMATRIX baseViewMatrix;

char videoCard[128];

int videoMemory;

// Create the input object. The input object will be used to handle reading the keyboard and mouse input from the user.

m\_Input = new InputClass;

if(!m\_Input)

{

return false;

}

// Initialize the input object.

result = m\_Input->Initialize(hinstance, hwnd, screenWidth, screenHeight);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the input object.", L"Error", MB\_OK);

return false;

}

// Create the Direct3D object.

m\_Direct3D = new D3DClass;

if(!m\_Direct3D)

{

return false;

}

// Initialize the Direct3D object.

result = m\_Direct3D->Initialize(screenWidth, screenHeight, VSYNC\_ENABLED, hwnd, FULL\_SCREEN, SCREEN\_DEPTH, SCREEN\_NEAR);

if(!result)

{

MessageBox(hwnd, L"Could not initialize DirectX 11.", L"Error", MB\_OK);

return false;

}

// Create the camera object.

m\_Camera = new CameraClass;

if(!m\_Camera)

{

return false;

}

// Initialize a base view matrix with the camera for 2D user interface rendering.

m\_Camera->SetPosition(0.0f, 0.0f, -1.0f);

m\_Camera->Render();

m\_Camera->GetViewMatrix(baseViewMatrix);

// Set the initial position of the camera.

cameraX = 150.0f;

cameraY = 2.0f;

cameraZ = 35.0f;

m\_Camera->SetPosition(cameraX, cameraY, cameraZ);

// Create the terrain object.

m\_Terrain = new TerrainClass;

if(!m\_Terrain)

{

return false;

}

The terrain object takes the names of the three new textures as input.

// Initialize the terrain object.

result = m\_Terrain->Initialize(m\_Direct3D->GetDevice(), "../Engine/data/heightmap.bmp", L"../Engine/data/grass.dds", L"../Engine/data/slope.dds",

L"../Engine/data/rock.dds");

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain object.", L"Error", MB\_OK);

return false;

}

// Create the timer object.

m\_Timer = new TimerClass;

if(!m\_Timer)

{

return false;

}

// Initialize the timer object.

result = m\_Timer->Initialize();

if(!result)

{

MessageBox(hwnd, L"Could not initialize the timer object.", L"Error", MB\_OK);

return false;

}

// Create the position object.

m\_Position = new PositionClass;

if(!m\_Position)

{

return false;

}

// Set the initial position of the viewer to the same as the initial camera position.

m\_Position->SetPosition(cameraX, cameraY, cameraZ);

// Create the fps object.

m\_Fps = new FpsClass;

if(!m\_Fps)

{

return false;

}

// Initialize the fps object.

m\_Fps->Initialize();

// Create the cpu object.

m\_Cpu = new CpuClass;

if(!m\_Cpu)

{

return false;

}

// Initialize the cpu object.

m\_Cpu->Initialize();

// Create the font shader object.

m\_FontShader = new FontShaderClass;

if(!m\_FontShader)

{

return false;

}

// Initialize the font shader object.

result = m\_FontShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the font shader object.", L"Error", MB\_OK);

return false;

}

// Create the text object.

m\_Text = new TextClass;

if(!m\_Text)

{

return false;

}

// Initialize the text object.

result = m\_Text->Initialize(m\_Direct3D->GetDevice(), m\_Direct3D->GetDeviceContext(), hwnd, screenWidth, screenHeight, baseViewMatrix);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the text object.", L"Error", MB\_OK);

return false;

}

// Retrieve the video card information.

m\_Direct3D->GetVideoCardInfo(videoCard, videoMemory);

// Set the video card information in the text object.

result = m\_Text->SetVideoCardInfo(videoCard, videoMemory, m\_Direct3D->GetDeviceContext());

if(!result)

{

MessageBox(hwnd, L"Could not set video card info in the text object.", L"Error", MB\_OK);

return false;

}

// Create the terrain shader object.

m\_TerrainShader = new TerrainShaderClass;

if(!m\_TerrainShader)

{

return false;

}

// Initialize the terrain shader object.

result = m\_TerrainShader->Initialize(m\_Direct3D->GetDevice(), hwnd);

if(!result)

{

MessageBox(hwnd, L"Could not initialize the terrain shader object.", L"Error", MB\_OK);

return false;

}

// Create the light object.

m\_Light = new LightClass;

if(!m\_Light)

{

return false;

}

// Initialize the light object.

m\_Light->SetAmbientColor(0.05f, 0.05f, 0.05f, 1.0f);

m\_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);

m\_Light->SetDirection(-0.5f, -1.0f, 0.0f);

return true;

}

void ApplicationClass::Shutdown()

{

// Release the light object.

if(m\_Light)

{

delete m\_Light;

m\_Light = 0;

}

// Release the terrain shader object.

if(m\_TerrainShader)

{

m\_TerrainShader->Shutdown();

delete m\_TerrainShader;

m\_TerrainShader = 0;

}

// Release the text object.

if(m\_Text)

{

m\_Text->Shutdown();

delete m\_Text;

m\_Text = 0;

}

// Release the font shader object.

if(m\_FontShader)

{

m\_FontShader->Shutdown();

delete m\_FontShader;

m\_FontShader = 0;

}

// Release the cpu object.

if(m\_Cpu)

{

m\_Cpu->Shutdown();

delete m\_Cpu;

m\_Cpu = 0;

}

// Release the fps object.

if(m\_Fps)

{

delete m\_Fps;

m\_Fps = 0;

}

// Release the position object.

if(m\_Position)

{

delete m\_Position;

m\_Position = 0;

}

// Release the timer object.

if(m\_Timer)

{

delete m\_Timer;

m\_Timer = 0;

}

// Release the terrain object.

if(m\_Terrain)

{

m\_Terrain->Shutdown();

delete m\_Terrain;

m\_Terrain = 0;

}

// Release the camera object.

if(m\_Camera)

{

delete m\_Camera;

m\_Camera = 0;

}

// Release the Direct3D object.

if(m\_Direct3D)

{

m\_Direct3D->Shutdown();

delete m\_Direct3D;

m\_Direct3D = 0;

}

// Release the input object.

if(m\_Input)

{

m\_Input->Shutdown();

delete m\_Input;

m\_Input = 0;

}

return;

}

bool ApplicationClass::Frame()

{

bool result;

// Read the user input.

result = m\_Input->Frame();

if(!result)

{

return false;

}

// Check if the user pressed escape and wants to exit the application.

if(m\_Input->IsEscapePressed() == true)

{

return false;

}

// Update the system stats.

m\_Timer->Frame();

m\_Fps->Frame();

m\_Cpu->Frame();

// Update the FPS value in the text object.

result = m\_Text->SetFps(m\_Fps->GetFps(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the CPU usage value in the text object.

result = m\_Text->SetCpu(m\_Cpu->GetCpuPercentage(), m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Do the frame input processing.

result = HandleInput(m\_Timer->GetTime());

if(!result)

{

return false;

}

// Render the graphics.

result = RenderGraphics();

if(!result)

{

return false;

}

return result;

}

bool ApplicationClass::HandleInput(float frameTime)

{

bool keyDown, result;

float posX, posY, posZ, rotX, rotY, rotZ;

// Set the frame time for calculating the updated position.

m\_Position->SetFrameTime(frameTime);

// Handle the input.

keyDown = m\_Input->IsLeftPressed();

m\_Position->TurnLeft(keyDown);

keyDown = m\_Input->IsRightPressed();

m\_Position->TurnRight(keyDown);

keyDown = m\_Input->IsUpPressed();

m\_Position->MoveForward(keyDown);

keyDown = m\_Input->IsDownPressed();

m\_Position->MoveBackward(keyDown);

keyDown = m\_Input->IsAPressed();

m\_Position->MoveUpward(keyDown);

keyDown = m\_Input->IsZPressed();

m\_Position->MoveDownward(keyDown);

keyDown = m\_Input->IsPgUpPressed();

m\_Position->LookUpward(keyDown);

keyDown = m\_Input->IsPgDownPressed();

m\_Position->LookDownward(keyDown);

// Get the view point position/rotation.

m\_Position->GetPosition(posX, posY, posZ);

m\_Position->GetRotation(rotX, rotY, rotZ);

// Set the position of the camera.

m\_Camera->SetPosition(posX, posY, posZ);

m\_Camera->SetRotation(rotX, rotY, rotZ);

// Update the position values in the text object.

result = m\_Text->SetCameraPosition(posX, posY, posZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

// Update the rotation values in the text object.

result = m\_Text->SetCameraRotation(rotX, rotY, rotZ, m\_Direct3D->GetDeviceContext());

if(!result)

{

return false;

}

return true;

}

bool ApplicationClass::RenderGraphics()

{

D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix;

bool result;

// Clear the scene.

m\_Direct3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

// Generate the view matrix based on the camera's position.

m\_Camera->Render();

// Get the world, view, projection, and ortho matrices from the camera and Direct3D objects.

m\_Direct3D->GetWorldMatrix(worldMatrix);

m\_Camera->GetViewMatrix(viewMatrix);

m\_Direct3D->GetProjectionMatrix(projectionMatrix);

m\_Direct3D->GetOrthoMatrix(orthoMatrix);

// Render the terrain buffers.

m\_Terrain->Render(m\_Direct3D->GetDeviceContext());

When we render the terrain now we supply it the three new terrain textures that will be mapped procedurally according to the terrain slope.

// Render the terrain using the terrain shader.

result = m\_TerrainShader->Render(m\_Direct3D->GetDeviceContext(), m\_Terrain->GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix,

m\_Light->GetAmbientColor(), m\_Light->GetDiffuseColor(), m\_Light->GetDirection(), m\_Terrain->GetGrassTexture(),

m\_Terrain->GetSlopeTexture(), m\_Terrain->GetRockTexture());

if(!result)

{

return false;

}

// Turn off the Z buffer to begin all 2D rendering.

m\_Direct3D->TurnZBufferOff();

// Turn on the alpha blending before rendering the text.

m\_Direct3D->TurnOnAlphaBlending();

// Render the text user interface elements.

result = m\_Text->Render(m\_Direct3D->GetDeviceContext(), m\_FontShader, worldMatrix, orthoMatrix);

if(!result)

{

return false;

}

// Turn off alpha blending after rendering the text.

m\_Direct3D->TurnOffAlphaBlending();

// Turn the Z buffer back on now that all 2D rendering has completed.

m\_Direct3D->TurnZBufferOn();

// Present the rendered scene to the screen.

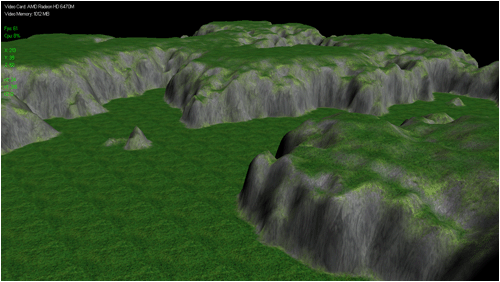
m\_Direct3D->EndScene();

return true;

}

Summary

We now have terrain that can be automatically texture mapped per pixel based on the slope.



To Do Exercises

1. Compile and run the program. Move around the terrain to see the effect of textures mapped based on slope.

2. Change the textures to create a different looking terrain.

3. Modify the slope values in the pixel shader to see the effect. You may want to change the textures to be pure colors like red, green, blue.

4. Add a fourth range to the slope calculate in the pixel shader and add another texture.