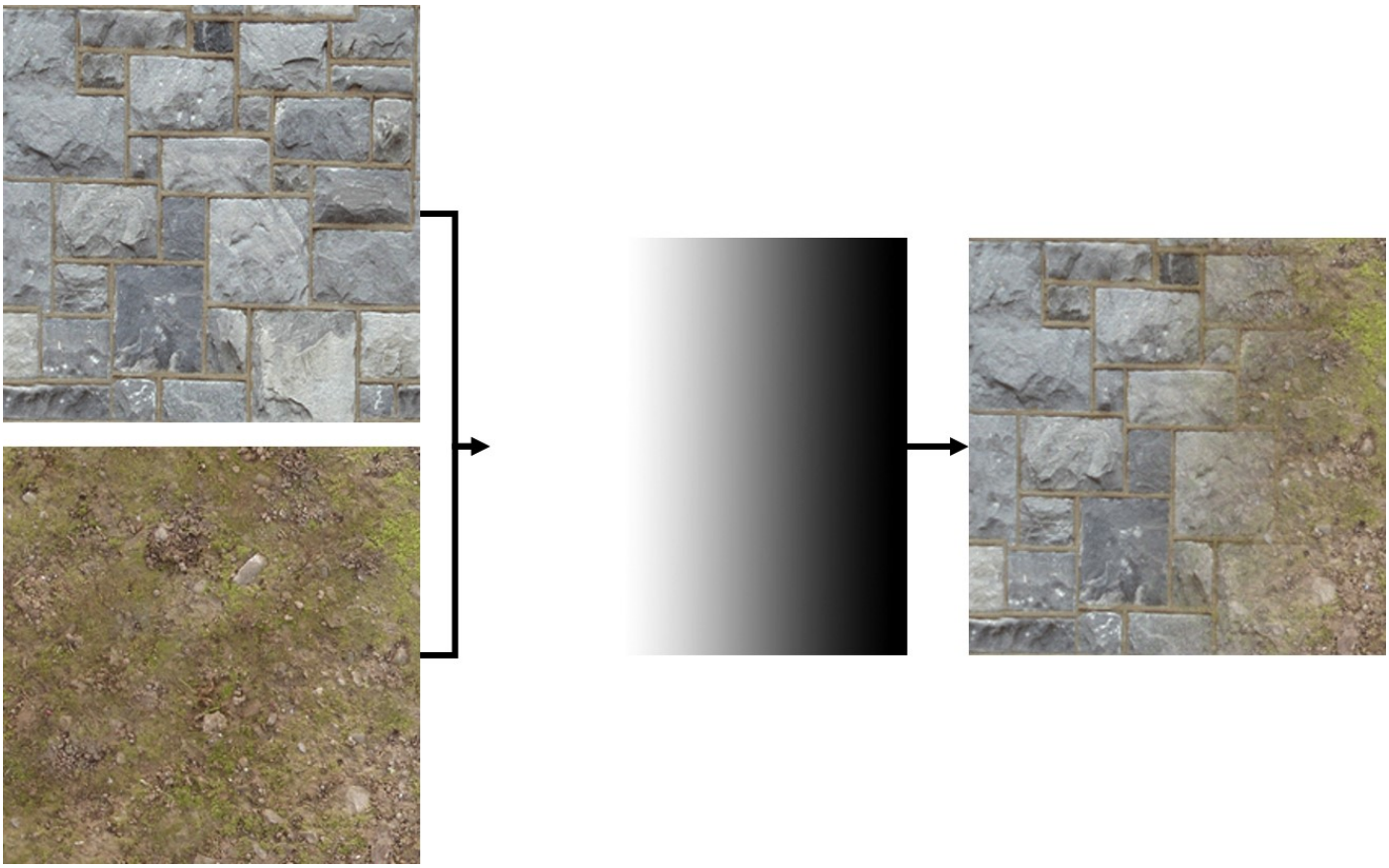
# Tutorial: Alpha Mapping

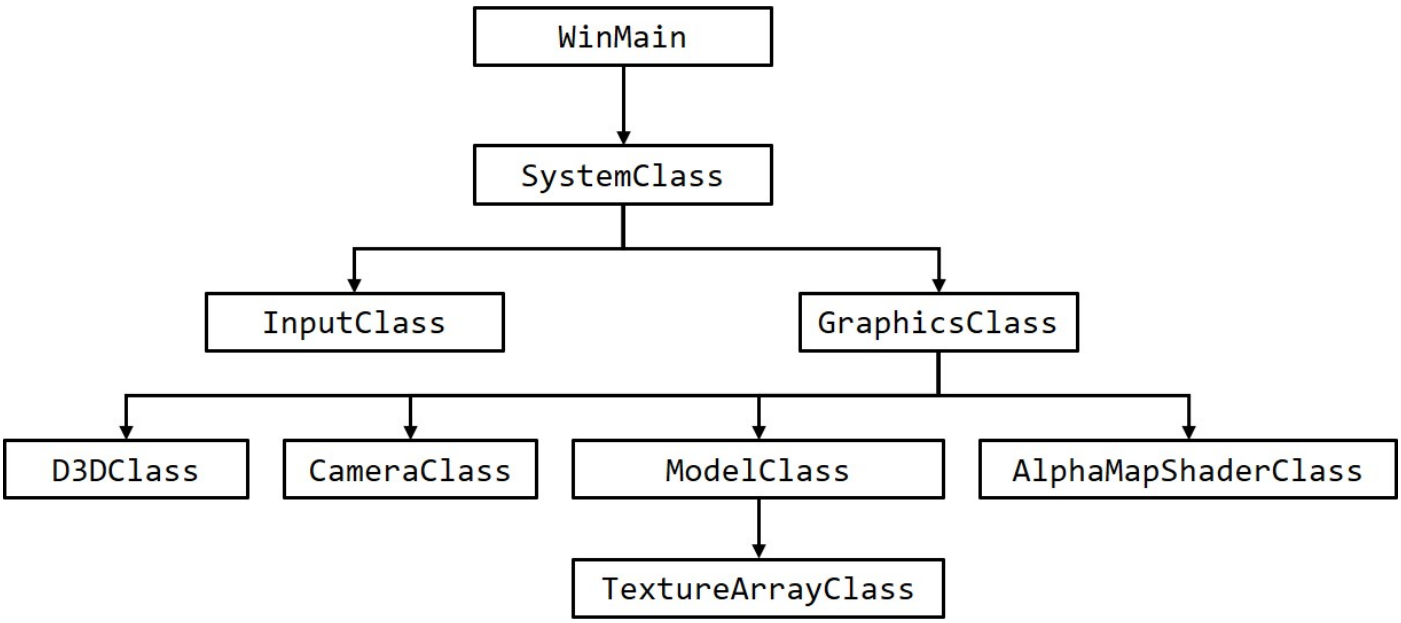
Alpha mapping in DirectX 11 is the process of using the alpha layer of a texture to determine the blending amount for each pixel when combining two textures.

Each pixel is just a 0.0 to 1.0 float range indicating how to combine two textures. For example if the alpha value at a certain pixel is 0.3 you would take 30% of the base texture pixel value and combine it with 70% of the color texture pixel. In this tutorial, the following two textures are combined based on the alpha map:



In this tutorial I will be separating the alpha map onto its own individual texture. This gives us the ability to create several alpha maps and then combine the same two color and base textures in many different ways.

## Framework



## Alphamap.vs

The alpha map vertex shader is just the light map shader renamed from the previous tutorial.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: alphamap.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 AlphaMapVertexShader(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;
}
```

## Alphamap.ps

```
////////////////////////////////////////////////////////////////////////////////
// Filename: alphamap.ps
////////////////////////////////////////////////////////////////////////////////

/////////////
// GLOBALS //
/////////////
```

The first change to the pixel shader is the addition of a third element in the texture array for holding the alpha map texture.

```
Texture2D shaderTextures[3];
SamplerState SampleType;

//////////////
// TYPEDEFS //
//////////////
struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
};
```

In the alpha map pixel shader we first take a sample of the pixel from the two color textures and alpha texture. Then we multiply the alpha value by the base color to get the pixel value for the base texture. After that we multiply the inverse of the alpha (1.0 - alpha) by the second color texture to get the pixel value for the second texture. We then add the two pixel values together and saturate to produce the final blended pixel.

```
////////////////////////////////////////////////////////////////////////////////
// Pixel Shader
////////////////////////////////////////////////////////////////////////////////
float4 AlphaMapPixelShader(PixelInputType input) : SV_TARGET
{
    float4 color1;
    float4 color2;
    float4 alphaValue;
    float4 blendColor;

    // Get the pixel color from the first texture.
    color1 = shaderTextures[0].Sample(SampleType, input.tex);

    // Get the pixel color from the second texture.
    color2 = shaderTextures[1].Sample(SampleType, input.tex);

    // Get the alpha value from the alpha map texture.
    alphaValue = shaderTextures[2].Sample(SampleType, input.tex);

    // Combine the two textures based on the alpha value.
    blendColor = (alphaValue * color1) + ((1.0 - alphaValue) * color2);

    // Saturate the final color value.
    blendColor = saturate(blendColor);

    return blendColor;
}
```

## Alphamapshaderclass.h

The AlphaMapShaderClass is the LightMapShaderClass slightly modified from the previous tutorial.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: alphamapshaderclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _ALPHAMAPSHADERCLASS_H_
#define _ALPHAMAPSHADERCLASS_H_

//////////////
// INCLUDES //
//////////////
#include <d3d11.h>
#include <d3dx10math.h>
#include <d3dx11async.h>
#include <fstream>
```

```
using namespace std;

////////////////////////////////////////////////////////////////////////////////
// Class name: AlphaMapShaderClass
////////////////////////////////////////////////////////////////////////////////
class AlphaMapShaderClass
{
private:
	struct MatrixBufferType
	{
		D3DXMATRIX world;
		D3DXMATRIX view;
		D3DXMATRIX projection;
	};

public:
	AlphaMapShaderClass();
	AlphaMapShaderClass(const AlphaMapShaderClass&);
	~AlphaMapShaderClass();

	bool Initialize(ID3D11Device*, HWND);
	void Shutdown();
	bool Render(ID3D11DeviceContext*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView**);

private:
	bool InitializeShader(ID3D11Device*, HWND, WCHAR*, WCHAR*);
	void ShutdownShader();
	void OutputShaderErrorMessage(ID3D10Blob*, HWND, WCHAR*);

	bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView**);
	void RenderShader(ID3D11DeviceContext*, int);

private:
	ID3D11VertexShader* m_vertexShader;
	ID3D11PixelShader* m_pixelShader;
	ID3D11InputLayout* m_layout;
	ID3D11Buffer* m_matrixBuffer;
	ID3D11SamplerState* m_sampleState;
};

#endif
```

## Alphamapshaderclass.cpp

```
////////////////////////////////////////////////////////////////////////////////
// Filename: alphamapshaderclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "alphamapshaderclass.h"

AlphaMapShaderClass::AlphaMapShaderClass()
{
	m_vertexShader = 0;
	m_pixelShader = 0;
	m_layout = 0;
	m_matrixBuffer = 0;
	m_sampleState = 0;
}

AlphaMapShaderClass::AlphaMapShaderClass(const AlphaMapShaderClass& other)
{
}

AlphaMapShaderClass::~AlphaMapShaderClass()
{
```

```
}

bool AlphaMapShaderClass::Initialize(ID3D11Device* device, HWND hwnd)
{
	bool result;
```

The first change is that the alphamap.vs and alphamap.ps HLSL shader files are now loaded.

```
	// Initialize the vertex and pixel shaders.
	result = InitializeShader(device, hwnd, L"../Engine/alphamap.vs", L"../Engine/alphamap.ps");
	if(!result)
	{
		return false;
	}

	return true;
}

void AlphaMapShaderClass::Shutdown()
{
	// Shutdown the vertex and pixel shaders as well as the related objects.
	ShutdownShader();

	return;
}

bool AlphaMapShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX
	worldMatrix, D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView**
	textureArray)
{
	bool result;


	// Set the shader parameters that it will use for rendering.
	result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, textureArray);
	if(!result)
	{
		return false;
	}

	// Now render the prepared buffers with the shader.
	RenderShader(deviceContext, indexCount);

	return true;
}

bool AlphaMapShaderClass::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_BUFFER_DESC matrixBufferDesc;
	D3D11_SAMPLER_DESC samplerDesc;

	// Initialize the pointers this function will use to null.
	errorMessage = 0;
	vertexShaderBuffer = 0;
	pixelShaderBuffer = 0;
```

The alpha map vertex shader is loaded here.

```
	// Compile the vertex shader code.
```

```
	result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "AlphaMapVertexShader", "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;
	}
```

The alpha map pixel shader is loaded here.

```
	// Compile the pixel shader code.
	result = D3DX11CompileFromFile(psFilename, NULL, NULL, "AlphaMapPixelShader", "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.
	// This setup needs to match the VertexType stucture in the ModelClass and in the shader.
	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;

	// 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;
	}

	// 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;
	}

	return true;
}
```

```
void AlphaMapShaderClass::ShutdownShader()
{
	// Release the sampler state.
	if(m_sampleState)
	{
		m_sampleState->Release();
		m_sampleState = 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 AlphaMapShaderClass::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 AlphaMapShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX
	worldMatrix, D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView**
	textureArray)
{
	HRESULT result;
	D3D11_MAPPED_SUBRESOURCE mappedResource;
	MatrixBufferType* dataPtr;
	unsigned int bufferNumber;


	// Transpose the matrices to prepare them for the shader.
	D3DXMatrixTranspose(&worldMatrix, &worldMatrix);
	D3DXMatrixTranspose(&viewMatrix, &viewMatrix);
	D3DXMatrixTranspose(&projectionMatrix, &projectionMatrix);

	// 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;

	// 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);
```

The next major change is that we now set three textures in the shader instead of two like in the previous tutorials.

```
	// Set shader texture array resource in the pixel shader.
	deviceContext->PSSetShaderResources(0, 3, textureArray);

	return true;
}


void AlphaMapShaderClass::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;
}
```

## Texturearrayclass.h

The TextureArrayClass has been changed to handle three textures instead of two.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: texturearrayclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _TEXTUREARRAYCLASS_H_
#define _TEXTUREARRAYCLASS_H_

//////////////
// INCLUDES //
//////////////
#include <d3d11.h>
#include <d3dx11tex.h>

////////////////////////////////////////////////////////////////////////////////
// Class name: TextureArrayClass
////////////////////////////////////////////////////////////////////////////////
class TextureArrayClass
{
public:
	TextureArrayClass();
	TextureArrayClass(const TextureArrayClass&);
	~TextureArrayClass();

	bool Initialize(ID3D11Device*, WCHAR*, WCHAR*, WCHAR*);
	void Shutdown();

	ID3D11ShaderResourceView** GetTextureArray();

private:
```

The number of elements in the texture array is changed to three.

```
	ID3D11ShaderResourceView* m_textures[3];
};

#endif
```

## Texturearrayclass.cpp

```
////////////////////////////////////////////////////////////////////////////////
// Filename: texturearrayclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "texturearrayclass.h"
```

The three textures are initialized to null in the class constructor.

```
TextureArrayClass::TextureArrayClass()
{
	m_textures[0] = 0;
	m_textures[1] = 0;
	m_textures[2] = 0;
```

```
}

TextureArrayClass::TextureArrayClass(const TextureArrayClass& other)
{
}

TextureArrayClass::~TextureArrayClass()
{
}
```

The Initialize function now loads three textures into the texture array.

```
bool TextureArrayClass::Initialize(ID3D11Device* device, WCHAR* filename1, WCHAR* filename2, WCHAR*
        filename3)
{
        HRESULT result;


        // Load the first texture in.
        result = D3DX11CreateShaderResourceViewFromFile(device, filename1, NULL, NULL, &m_textures[0],
                NULL);
        if(FAILED(result))
        {
                return false;
        }

        // Load the second texture in.
        result = D3DX11CreateShaderResourceViewFromFile(device, filename2, NULL, NULL, &m_textures[1],
                NULL);
        if(FAILED(result))
        {
                return false;
        }

        // Load the third texture in.
        result = D3DX11CreateShaderResourceViewFromFile(device, filename3, NULL, NULL, &m_textures[2],
                NULL);
        if(FAILED(result))
        {
                return false;
        }

        return true;
}
```

Shutdown now releases three textures.

```
void TextureArrayClass::Shutdown()
{
        // Release the texture resources.
        if(m_textures[0])
        {
                m_textures[0]->Release();
                m_textures[0] = 0;
        }

        if(m_textures[1])
        {
                m_textures[1]->Release();
                m_textures[1] = 0;
        }

        if(m_textures[2])
        {
                m_textures[2]->Release();
                m_textures[2] = 0;
```

```
	}

	return;
}

ID3D11ShaderResourceView** TextureArrayClass::GetTextureArray()
{
	return m_textures;
}
```

## Modelclass.h

The ModelClass has been modified just slightly to handle three textures instead of two.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: modelclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _MODELCLASS_H_
#define _MODELCLASS_H_

//////////////
// INCLUDES //
//////////////
#include <d3d11.h>
#include <d3dx10math.h>
#include <fstream>
using namespace std;

///////////////////////
// MY CLASS INCLUDES //
///////////////////////
#include "texturearrayclass.h"

////////////////////////////////////////////////////////////////////////////////
// Class name: ModelClass
////////////////////////////////////////////////////////////////////////////////
class ModelClass
{
private:
	struct VertexType
	{
		D3DXVECTOR3 position;
		D3DXVECTOR2 texture;
	};

	struct ModelType
	{
		float x, y, z;
		float tu, tv;
		float nx, ny, nz;
	};

public:
	ModelClass();
	ModelClass(const ModelClass&);
	~ModelClass();

	bool Initialize(ID3D11Device*, char*, WCHAR*, WCHAR*, WCHAR*);
	void Shutdown();
	void Render(ID3D11DeviceContext*);

	int GetIndexCount();
	ID3D11ShaderResourceView** GetTextureArray();

private:
	bool InitializeBuffers(ID3D11Device*);
```

```
        void ShutdownBuffers();
        void RenderBuffers(ID3D11DeviceContext*);

        bool LoadTextures(ID3D11Device*, WCHAR*, WCHAR*, WCHAR*);
        void ReleaseTextures();

        bool LoadModel(char*);
        void ReleaseModel();

private:
        ID3D11Buffer *m_vertexBuffer, *m_indexBuffer;
        int m_vertexCount, m_indexCount;
        ModelType* m_model;
        TextureArrayClass* m_TextureArray;
};

#endif
```

## Modelclass.cpp

I will only cover the functions that have changed since the previous tutorial.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: modelclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "modelclass.h"
```

The Initialize function now takes as input three texture names. The first two are the color texture and the third is the alpha texture.

```
bool ModelClass::Initialize(ID3D11Device* device, char* modelFilename, WCHAR* textureFilename1, WCHAR*
textureFilename2,
                            WCHAR* textureFilename3)
{
        bool result;

        // Load in the model data,
        result = LoadModel(modelFilename);
        if(!result)
        {
                return false;
        }

        // Initialize the vertex and index buffers.
        result = InitializeBuffers(device);
        if(!result)
        {
                return false;
        }
```

LoadTextures now takes the three texture names as input.

```
        // Load the textures for this model.
        result = LoadTextures(device, textureFilename1, textureFilename2, textureFilename3);
        if(!result)
        {
                return false;
        }

        return true;
}
```

The LoadTextures function now takes the three texture file names as input and then creates and loads a texture array using the three texture files. Once again the first two textures are the color textures and the third is the alpha texture.

```
bool ModelClass::LoadTextures(ID3D11Device* device, WCHAR* filename1, WCHAR* filename2, WCHAR*
	filename3)
{
	bool result;


	// Create the texture array object.
	m_TextureArray = new TextureArrayClass;
	if(!m_TextureArray)
	{
		return false;
	}

	// Initialize the texture array object.
	result = m_TextureArray->Initialize(device, filename1, filename2, filename3);
	if(!result)
	{
		return false;
	}

	return true;
}
```

## Graphicsclass.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: graphicsclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _GRAPHICSCLASS_H_
#define _GRAPHICSCLASS_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 "d3dclass.h"
#include "cameraclass.h"
#include "modelclass.h"
```

The new AlphaMapShaderClass header is now included in the GraphicsClass header file.

```
#include "alphamapshaderclass.h"

////////////////////////////////////////////////////////////////////////////////
// Class name: GraphicsClass
////////////////////////////////////////////////////////////////////////////////
class GraphicsClass
{
public:
	GraphicsClass();
	GraphicsClass(const GraphicsClass&);
	~GraphicsClass();

	bool Initialize(int, int, HWND);
	void Shutdown();
	bool Frame();
	bool Render();
```

```
private:
	D3DClass* m_D3D;
	CameraClass* m_Camera;
	ModelClass* m_Model;
```

We create the new AlphaMapShaderClass object here.

```
	AlphaMapShaderClass* m_AlphaMapShader;
};

#endif
```

## Graphicsclass.cpp

I will only cover the functions that have changed since the previous tutorial.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: graphicsclass.cpp
////////////////////////////////////////////////////////////////////////////////
#include "graphicsclass.h"

GraphicsClass::GraphicsClass()
{
	m_D3D = 0;
	m_Camera = 0;
	m_Model = 0;
```

The new AlphaMapShaderClass object is initialized to null in the class constructor.

```
	m_AlphaMapShader = 0;
}

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
	bool result;
	D3DXMATRIX baseViewMatrix;


	// Create the Direct3D object.
	m_D3D = new D3DClass;
	if(!m_D3D)
	{
		return false;
	}

	// Initialize the Direct3D object.
	result = m_D3D->Initialize(screenWidth, screenHeight, VSYNC_ENABLED, hwnd, FULL_SCREEN,
		SCREEN_DEPTH, SCREEN_NEAR);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize Direct3D", 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);

	// Create the model object.
	m_Model = new ModelClass;
	if(!m_Model)
	{
		return false;
	}
```

The ModelClass object is initialized with three textures. The first two textures are the color textures. The third input texture is the alpha texture that will be used to blend the first two textures.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), "../Engine/data/square.txt",
		L"../Engine/data/stone01.dds", L"../Engine/data/dirt01.dds", L"../Engine/data/alpha01.dds");
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB_OK);
		return false;
	}
```

The new AlphaMapShaderClass object is created and initialized here.

```
	// Create the alpha map shader object.
	m_AlphaMapShader = new AlphaMapShaderClass;
	if(!m_AlphaMapShader)
	{
		return false;
	}

	// Initialize the alpha map shader object.
	result = m_AlphaMapShader->Initialize(m_D3D->GetDevice(), hwnd);
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the alpha map shader object.", L"Error", MB_OK);
		return false;
	}

	return true;
}

void GraphicsClass::Shutdown()
{
```

The new AlphaMapShaderClass object is released here in the Shutdown function.

```
	// Release the alpha map shader object.
	if(m_AlphaMapShader)
	{
		m_AlphaMapShader->Shutdown();
		delete m_AlphaMapShader;
		m_AlphaMapShader = 0;
	}

	// Release the model object.
	if(m_Model)
	{
		m_Model->Shutdown();
		delete m_Model;
		m_Model = 0;
	}
```

```
	// Release the camera object.
	if(m_Camera)
	{
		delete m_Camera;
		m_Camera = 0;
	}

	// Release the D3D object.
	if(m_D3D)
	{
		m_D3D->Shutdown();
		delete m_D3D;
		m_D3D = 0;
	}

	return;
}

bool GraphicsClass::Render()
{
	D3DXMATRIX worldMatrix, viewMatrix, projectionMatrix, orthoMatrix;


	// Clear the buffers to begin the scene.
	m_D3D->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 D3D objects.
	m_D3D->GetWorldMatrix(worldMatrix);
	m_Camera->GetViewMatrix(viewMatrix);
	m_D3D->GetProjectionMatrix(projectionMatrix);
	m_D3D->GetOrthoMatrix(orthoMatrix);

	// Put the model vertex and index buffers on the graphics pipeline to prepare them for drawing.
	m_Model->Render(m_D3D->GetDeviceContext());
```

The new AlphaMapShaderClass object is used to render the model object using alpha blending.

```
	// Render the model using the alpha map shader.
	m_AlphaMapShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
		viewMatrix, projectionMatrix,	m_Model->GetTextureArray());

	// Present the rendered scene to the screen.
	m_D3D->EndScene();

	return true;
}
```

## Summary

Alpha mapping provides an easy way of controlling on a very fine level of how to combine textures. Many terrain based applications use this to provide smooth transitions between different textures over a very large landscape.

## To Do Exercises

1. Recompile and run the program to see the alpha mapped texture combination. Press escape to quit.

2. Make some of your own alpha maps and use them to combine the two textures in different ways.

*[Original script: www.rastertek.com]*