Tutorial: Multi-texturing

This tutorial will cover how to do multitexturing in DirectX 11 as well as how to implement texture arrays in DirectX 11. Multitexturing is the process of blending two different textures to create a final texture. The equation you use to blend the two textures can differ depending on the result you are trying to achieve. In this tutorial we will look at just combining the average pixel color of the two textures to create an evenly blended final texture.

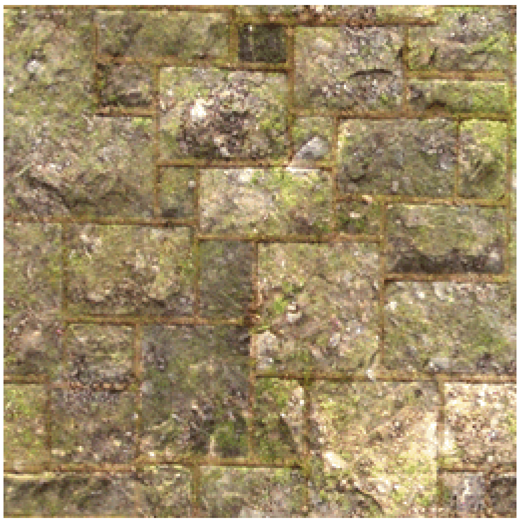
Texture arrays is a new feature since DirectX 10 that allows you to have multiple textures active at once in the GPU. Past methods where only a single texture was ever active in the GPU caused a lot of extra processing to load and unload textures all the time. Most people got around this problem by using texture aliases (loading a bunch of textures onto one large texture) and just using different UV coordinates. However, texture aliases are no longer needed with this new feature.

The first (base) and second (color) textures used in this tutorial look like the following:

These two textures will be combined in the pixel shader on a pixel by pixel basis. The blending equation we will use will be the following:

```
blendColor = basePixel * colorPixel * gammaCorrection;
```

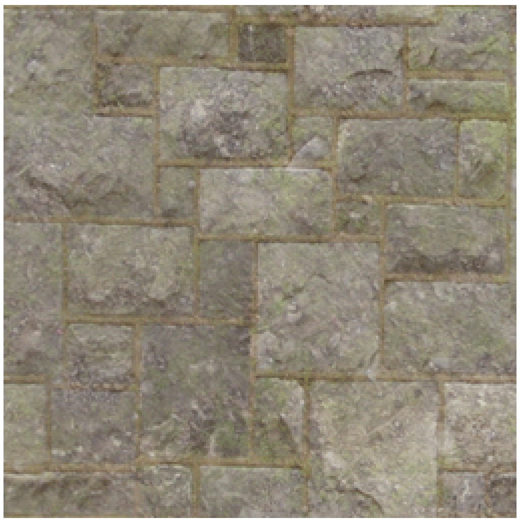
Using that equation and the two textures we will get the following result:



Now you may be wondering why I didn't just add together the average of the pixels such as the following:

```
blendColor = (basePixel * 0.5) + (colorPixel * 0.5);
```

The reason being is that the pixel color presented to us has been corrected to the gamma of the monitor. This makes the pixel values from 0.0 to 1.0 follow a non-linear curve. Therefore, we need gamma correction when working in the pixel shader to deal with non-linear color values. If we don't correct for gamma and just do the average addition function, we get a washed-out result such as this:



Also note that most devices have different gamma values and most require a look up table or a gamma slider so the user can choose the gamma settings for their device. In this example I just choose 2.0 as my gamma value to make the tutorial simple.

We'll start the code section by first looking at the new multitexture shader which was originally based on the texture shader file with some slight changes. All changes are made from Tutorial 5: Texturing.

Multitexture.vs

The only change to the vertex shader is the name.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: multitexture.vs
////////////////////////////////////////////////////////////////////////////////
```

[Same as previous codes]

```
////////////////////////////////////////////////////////////////////////////////
// Vertex Shader
////////////////////////////////////////////////////////////////////////////////
PixelInputType MultiTextureVertexShader(VertexInputType input)
{
```

[Same as previous codes]

```
}
```

Multitexture.ps

```
////////////////////////////////////////////////////////////////////////////////
// Filename: multitexture.ps
////////////////////////////////////////////////////////////////////////////////

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

We have added a two element texture array resource here for the two different textures that will be blended together. Texture arrays are more efficient that using single texture resources in terms of performance on the graphics card. Switching textures was very costly in earlier versions of DirectX forcing most engines to be written around texture and material switches. Texture arrays help reduce that performance cost.

```
Texture2D shaderTextures[2];
```

```
SamplerState SampleType;


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

The pixel shader is where all the work of this tutorial is done. We take a sample of the pixel from both textures at this current texture coordinate. After that we combine them using multiplication since they are non-linear due to gamma correction. We also multiply by a gamma value, we have used 2.0 in this example as it is close to most monitor's gamma value. Once we have the blended pixel we saturate it and then return it as our final result. Notice also the indexing method used to access the two textures in the texture array.

```
////////////////////////////////////////////////////////////////////////////////
// Pixel Shader
////////////////////////////////////////////////////////////////////////////////
float4 MultiTexturePixelShader(PixelInputType input) : SV_TARGET
{
	float4 color1;
	float4 color2;
	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);

	// Blend the two pixels together and multiply by the gamma value.
	blendColor = color1 * color2 * 2.0;

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

	return blendColor;
}
```

Multitextureshaderclass.h

The multitexture shader code is based on the TextureShaderClass with some slight modifications.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: multitextureshaderclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _MULTITEXTURESHADERCLASS_H_
#define _MULTITEXTURESHADERCLASS_H_
```

[Same as previous codes]

```
////////////////////////////////////////////////////////////////////////////////
// Class name: MultiTextureShaderClass
////////////////////////////////////////////////////////////////////////////////
class MultiTextureShaderClass
{
private:
```

[Same as previous codes]

```
public:
		MultiTextureShaderClass();
		MultiTextureShaderClass(const MultiTextureShaderClass&);
		~MultiTextureShaderClass();

					[Same as previous codes]

private:

					[Same as previous codes]

};

#endif
```

Multitextureshaderclass.cpp

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


MultiTextureShaderClass::MultiTextureShaderClass()
{

					[Same as previous codes]

}

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

MultiTextureShaderClass::~MultiTextureShaderClass()
{
}

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

The multitexture HLSL shader files are loaded here in the Initialize function.

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

		return true;
}
```

Shutdown calls the ShutdownShader function to release the shader related interfaces.

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

		return;
}
```

The Render function now takes as input a pointer to the texture array. This will give the shader access to the two textures for blending operations.

```
bool MultiTextureShaderClass::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;
}
```

The InitializeShader function loads the vertex and pixel shader as well as setting up the layout, matrix buffer, and sample state.

```
bool MultiTextureShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR*
psFilename)
{
```

[Same as previous codes]

The multitexture vertex shader is loaded here.

```
	// Compile the vertex shader code.
	result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "MultiTextureVertexShader", "vs_5_0",
		D3D10_SHADER_ENABLE_STRICTNESS, 0, NULL, &vertexShaderBuffer, &errorMessage,
		NULL);
```

[Same as previous codes]

The multitexture pixel shader is loaded here.

```
	// Compile the pixel shader code.
	result = D3DX11CompileFromFile(psFilename, NULL, NULL, "MultiTexturePixelShader", "ps_5_0",
		D3D10_SHADER_ENABLE_STRICTNESS, 0, NULL, &pixelShaderBuffer, &errorMessage,
		NULL);
```

[Same as previous codes]

```
}
```

ShutdownShader releases all the interfaces that were setup in the InitializeShader function.

```
void MultiTextureShaderClass::ShutdownShader()
{
```

[Same as previous codes]

```
}
```

The OutputShaderErrorMessage function writes out an error to a file if there is an issue compiling the vertex or pixel shader HLSL files.

```
void MultiTextureShaderClass::OutputShaderErrorMessage(ID3D10Blob* errorMessage, HWND hwnd, WCHAR*
shaderFilename)
{

                                        [Same as previous codes]

}
```

SetShaderParameters sets the matrices and texture array in the shader before rendering.

```
bool MultiTextureShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX
worldMatrix, D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView** textureArray)
{

                                        [Same as previous codes]
```

Here is where the texture array is set before rendering. The PSSetShaderResources function is used to set the texture array. The first parameter is where to start in the array. The second parameter is how many textures are in the array that is being passed in. And the third parameter is a pointer to the texture array.

```
	// Set shader texture array resource in the pixel shader.
	deviceContext->PSSetShaderResources(0, 2, textureArray);

	return true;
}
```

The RenderShader function sets the layout, shaders, and sampler. It then draws the model using the shader.

```
void MultiTextureShaderClass::RenderShader(ID3D11DeviceContext* deviceContext, int indexCount)
{

                                        [Same as previous codes]

}
```

Texturearrayclass.h

The TextureArrayClass replaces the TextureClass that was used before. Instead of having just a single texture it can now have multiple textures and give calling objects access to those textures. For this tutorial it just handles two textures, but it can easily be expanded.

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

                                        [Same as previous codes]

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

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

	ID3D11ShaderResourceView** GetTextureArray();
```

private:

This is the two element texture array private variable.

```
	ID3D11ShaderResourceView* m_textures[2];
};
```

#endif

Texturearrayclass.cpp

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

The class constructor initializes the texture array elements to null.

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


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


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

Initialize takes in the two texture file names and creates two texture resources in the texture array from those files.

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

	return true;
}
```

The Shutdown function releases each element in the texture array.

```
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;
	}

	return;
}
```

GetTextureArray returns a pointer to the texture array so calling objects can have access to the textures in the texture array.

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

Modelclass.h

ModelClass has been modified to now use texture arrays.

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

//////////////
// INCLUDES //
//////////////
#include <d3d11.h>
#include <d3dx10math.h>

#include <fstream>
using namespace std;
```

The TextureArrayClass header file is included instead of the previous TextureClass header file.

```
///////////////////////
// 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*);
		void Shutdown();
		void Render(ID3D11DeviceContext*);

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

private:
		bool InitializeBuffers(ID3D11Device*);
		void ShutdownBuffers();
		void RenderBuffers(ID3D11DeviceContext*);

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

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

private:
		ID3D11Buffer *m_vertexBuffer, *m_indexBuffer;
		int m_vertexCount, m_indexCount;
		ModelType* m_model;
```

We now have a TextureArrayClass variable instead of a TextureClass variable.

```
		TextureArrayClass* m_TextureArray;
};

#endif
```

Modelclass.cpp

I will just cover the functions that have changed since the previous tutorials.

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


ModelClass::ModelClass()
{
		m_vertexBuffer = 0;
		m_indexBuffer = 0;
		m_model = 0;
```

We initialize the new TextureArray variable in the class constructor.

```
		m_TextureArray = 0;
}


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

We call the LoadTextures function which takes in multiple file names for textures that will be loaded into the texture array and rendered as a blended result on this model.

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

	return true;
}

void ModelClass::Shutdown()
{
```

ReleaseTextures is called to release the texture array in the Shutdown function.

```
	// Release the model textures.
	ReleaseTextures();

	// Shutdown the vertex and index buffers.
	ShutdownBuffers();

	// Release the model data.
	ReleaseModel();

	return;
}
```

We have a new function called GetTextureArray which gives calling objects access to the texture array that is used for rendering this model.

```
ID3D11ShaderResourceView** ModelClass::GetTextureArray()
{
	return m_TextureArray->GetTextureArray();
}
```

LoadTextures has been changed to create a TextureArrayClass object and then initialize it by loading in the two textures that are given as input to this function.

```
bool ModelClass::LoadTextures(ID3D11Device* device, WCHAR* filename1, WCHAR* filename2)
{
	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);
	if(!result)
	{
		return false;
	}

	return true;
}
```

The ReleaseTextures function releases the TextureArrayClass object.

```
void ModelClass::ReleaseTextures()
{
	// Release the texture array object.
	if(m_TextureArray)
	{
		m_TextureArray->Shutdown();
		delete m_TextureArray;
		m_TextureArray = 0;
	}

	return;
}
```

Graphicsclass.h

```
////////////////////////////////////////////////////////////////////////////////
// Filename: graphicsclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _GRAPHICSCLASS_H_
#define _GRAPHICSCLASS_H_

				[Same as previous codes]
```

The header for the MultiTextureShaderClass is now included in the GraphicsClass.

```
#include "multitextureshaderclass.h"

////////////////////////////////////////////////////////////////////////////////
// Class name: GraphicsClass
////////////////////////////////////////////////////////////////////////////////
class GraphicsClass
{
public:

				[Same as previous codes]

private:

				[Same as previous codes]
```

Here we create the new MultiTextureShaderClass object.

```
	MultiTextureShaderClass* m_MultiTextureShader;
};

#endif
```

Graphicsclass.cpp

We will cover just the functions that have changed since the previous tutorials.

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

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

We initialize the new multitexture shader object to null in the class constructor.

```
	m_MultiTextureShader = 0;
}

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

				[Same as previous codes]

	// 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 now initialized differently. For this tutorial we load in the square.txt model as the effect we want to display works best on just a plain square. We also now load in two textures for the texture array instead of just a single texture like we did in previous tutorials.

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

Here we create and initialize the new multitexture shader object.

```
	// Create the multitexture shader object.
	m_MultiTextureShader = new MultiTextureShaderClass;
	if(!m_MultiTextureShader)
	{
		return false;
	}

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

void GraphicsClass::Shutdown()
{
```

We release the multitexture shader in the Shutdown function.

```
	// Release the multitexture shader object.
	if(m_MultiTextureShader)
	{
		m_MultiTextureShader->Shutdown();
		delete m_MultiTextureShader;
		m_MultiTextureShader = 0;
	}
```

[Same as previous codes]

```
}

bool GraphicsClass::Frame()
{
```

We set the position of the camera a bit closer to see the blending effect more clearly.

```
	// Set the position of the camera.
	m_Camera->SetPosition(0.0f, 0.0f, -5.0f);

	return true;
}

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());
```

We use the new multitexture shader to render the model. Notice that we send in the texture array from the ModelClass as input to the shader.

```
	// Render the model using the multitexture shader.
	m_MultiTextureShader->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

We now have a shader that will combine two textures evenly and apply gamma correction. We also now know how to use texture arrays for improved graphics performance.

To Do Exercises

1. Recompile the code and run the program to see the resulting image. Press escape to quit.

2. Replace the two textures with two new ones to see the results.

[Original script: www.rastertek.com]