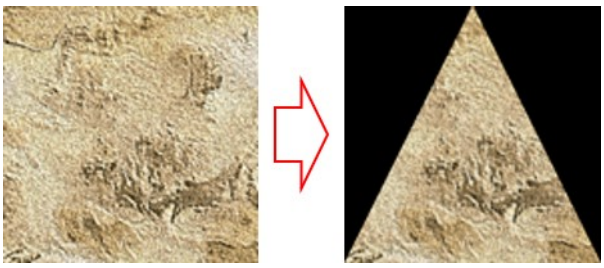
Tutorial: Texturing

[Original script: www.rastertek.com]

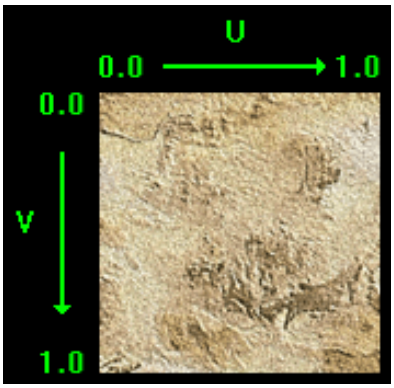
This tutorial will explain how to use **texturing** in DirectX 11. Texturing allows us to add photorealism to our scenes by applying photographs and other images onto polygon faces. For example, in this tutorial, we will take the following image and then apply it to the polygon from the previous tutorial to produce the following:



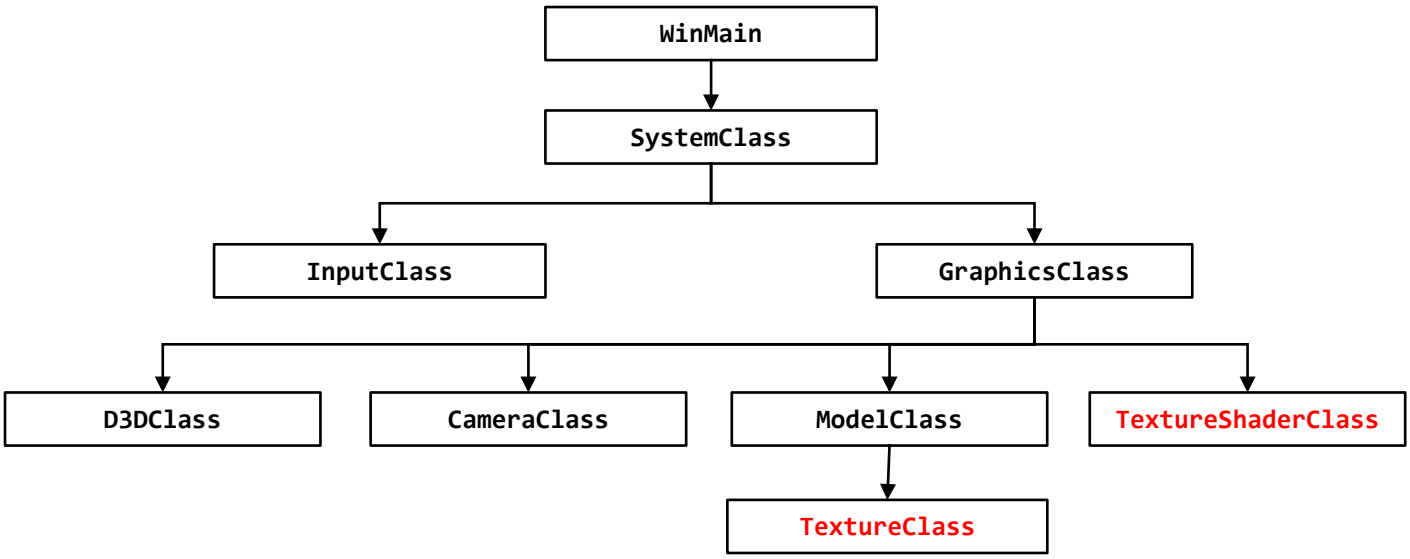
The format of the textures we will be using are **.dds** (Direct Draw Surface) files which are the format that DirectX uses. The tool used to produce .dds files comes with the DirectX SDK. It is under DirectX Utilities and is called DirectX Texture Tool. You can create a new texture of any size and format and then cut and paste your image or other format texture onto it and save it as a .dds file. It is very simple to use.

And before we get into the code, we should discuss how texture mapping works. To map pixels from the .dds image onto the polygon we use what is called the t**exel coordinate system**. This system converts the integer value of the pixel into a floating-point value between 0.0f and 1.0f. For example, if a texture width is 256 pixels wide then the first pixel will map to 0.0f, the 256th pixel will map to 1.0f, and a middle pixel of 128 would map to 0.5f.

In the texel coordinate system, the width value is named **"U"** and the height value is named **"V"**. The width goes from 0.0 on the left to 1.0 on the right. The height goes from 0.0 on the top to 1.0 on the bottom. For example, top left would be denoted as U 0.0, V 0.0 and bottom right would be denoted as U 1.0, V 1.0. we have made a diagram below to illustrate this system:



Framework



The changes to the framework since the previous tutorial is the new TextureClass which is inside ModelClass and the new TextureShaderClass which replaces the ColorShaderClass. We'll start the code section by looking at the new HLSL texture shaders first.

texture.vs

The texture vertex shader is similar to the previous color shader except that there have been some changes to accommodate texturing.

We are no longer using color in our vertex type and have instead moved to using texture coordinates. Since texture coordinates take a U and V float coordinate, we use float2 as its type. The semantic for texture coordinates is TEXCOORD0 for vertex shaders and pixel shaders. You can change the zero to any number to indicate which set of coordinates you are working with as multiple texture coordinates are allowed.

```
struct VertexInputType
{
    float4 position : POSITION;
    ~~float4 color : COLOR;~~
    float2 tex : TEXCOORD0;
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    ~~float4 color : COLOR;~~
    float2 tex : TEXCOORD0;
};

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

The only difference in the texture vertex shader in comparison to the color vertex shader from the previous tutorial is that instead of taking a copy of the color from the input vertex we take a copy of the texture coordinates and pass them to the pixel shader.

```
    ~~// Store the input color for the pixel shader to use.~~
    ~~output.color = input.color;~~
    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;

    return output;
}
```

texture.ps

The texture pixel shader has two global variables. The first is Texture2D shaderTexture which is the texture resource. This will be our texture resource that will be used for rendering the texture on the model. The second new variable is the SamplerState SampleType. The sampler state allows us to modify how the pixels are written to the polygon face when shaded. For example, if the polygon is really far away and only makes up 8 pixels on the screen then we use the sample state to figure out which pixels or what combination of pixels will actually be drawn from the original texture. The original texture may be 256 pixels by 256 pixels so deciding which pixels get drawn is important to ensure that the texture still looks decent on the really small polygon face. We will setup the sampler state in the TextureShaderClass also and then attach it to the resource pointer so this pixel shader can use it to determine which sample of pixels to draw.

```
// Globals
Texture2D shaderTexture;
SamplerState SampleType;
```

The PixelInputType for the texture pixel shader is also modified using texture coordinates instead of the color values.

```
// Type definitions
struct PixelInputType
{
    float4 position : SV_POSITION;
    float4 color : COLOR;
    float2 tex : TEXCOORD0;
};
```

The pixel shader has been modified so that it now uses the HLSL sample function. The sample function uses the sampler state we defined above and the texture coordinates for this pixel. It uses these two variables to determine and return the pixel value for this UV location on the polygon face.

```
// Pixel Shader
float4 ColorPixelShader(PixelInputType input) : SV_TARGET
float4 TexturePixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor;

    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    textureColor = shaderTexture.Sample(SampleType, input.tex);

    return input.color;
    return textureColor;
}
```

textureclass.h

The TextureClass encapsulates the loading, unloading, and accessing of a single texture resource. For each texture needed an object of this class must be instantiated.

```
#ifndef _TEXTURECLASS_H_
#define _TEXTURECLASS_H_

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

class TextureClass
{
public:
        TextureClass();
        TextureClass(const TextureClass&);
        ~TextureClass();
```

The first two functions will load a texture from a given file name and unload that texture when it is no longer needed.

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

The GetTexture function returns a pointer to the texture resource so that it can be used for rendering by shaders.

```
        ID3D11ShaderResourceView* GetTexture();

private:
```

This is the private texture resource.

```
        ID3D11ShaderResourceView* m_texture;
};

#endif
```

textureclass.cpp

```
#include "textureclass.h"
```

The class constructor will initialize the texture shader resource pointer to null.

```
TextureClass::TextureClass()
{
        m_texture = 0;
}

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

TextureClass::~TextureClass()
{
}
```

Initialize takes in the Direct3D device and file name of the texture and then loads the texture file into the shader resource variable called m_texture. The texture can now be used to render with.

```
bool TextureClass::Initialize(ID3D11Device* device, WCHAR* filename)
{
        HRESULT result;


        // Load the texture in.
        result = D3DX11CreateShaderResourceViewFromFile(device, filename, NULL, NULL, &m_texture, NULL);
        if(FAILED(result))
        {
                return false;
        }

        return true;
}
```

The Shutdown function releases the texture resource if it has been loaded and then sets the pointer to null.

```
void TextureClass::Shutdown()
{
        // Release the texture resource.
        if(m_texture)
        {
                m_texture->Release();
                m_texture = 0;
        }

        return;
}
```

GetTexture is the function that is called by other objects that need access to the texture shader resource so that they can use the texture for rendering.

```
ID3D11ShaderResourceView* TextureClass::GetTexture()
{
	return m_texture;
}
```

modelclass.h

The ModelClass has had changes since the previous tutorial so that it can now accommodate texturing. The TextureClass header is now included in the ModelClass header.

```
#include "textureclass.h"

class ModelClass
{
private:
```

The VertexType has replaced the color component with a texture coordinate component. The texture coordinate is now replacing the green color that was used in the previous tutorial.

```
	struct VertexType
	{
		D3DXVECTOR3 position;
		D3DXVECTOR4 color;
		D3DXVECTOR2 texture;
	};
```

The ModelClass also has a GetTexture function so it can pass its own texture resource to shaders that will draw this model.

```
public:
	bool Initialize(ID3D11Device*);
	bool Initialize(ID3D11Device*, WCHAR*);
	ID3D11ShaderResourceView* GetTexture();
```

ModelClass has both a private LoadTexture and ReleaseTexture for loading and releasing the texture that will be used to render this model.

```
private:
	bool LoadTexture(ID3D11Device*, WCHAR*);
	void ReleaseTexture();
```

The m_Texture variable is used for loading, releasing, and accessing the texture resource for this model.

```
private:
	TextureClass* m_Texture;
};
```

modelclass.cpp

```
ModelClass::ModelClass()
{
```

The class constructor now initializes the new texture object to null.

```
	m_Texture = 0;
}
```

Initialize now takes as input the file name of the .dds texture that the model will be using.

```
bool ModelClass::Initialize(ID3D11Device* device)
bool ModelClass::Initialize(ID3D11Device* device, WCHAR* textureFilename)
{
```

The Initialize function now calls a new private function that will load the texture.

```
	// Load the texture for this model.
	result = LoadTexture(device, textureFilename);
	if(!result)
	{
		return false;
	}

	return true;
}

void ModelClass::Shutdown()
{
```

The Shutdown function now calls a new private function to release the texture object that was loaded during initialization.

```
	// Release the vertex and index buffers.
	ShutdownBuffers();

	return true;
}
```

GetTexture returns the model texture resource. The texture shader will need access to this texture to render the model.

```
ID3D11ShaderResourceView* ModelClass::GetTexture()
{
	return m_Texture->GetTexture();
}

bool ModelClass::InitializeBuffers(ID3D11Device* device)
{
```

The vertex array now has a texture component instead of a color component. The texture vector is always U first and V second. For example the first texture coordinate is bottom left of the triangle which corresponds to U 0.0, V 1.0. Use the diagram at the top of this page to figure out what your coordinates need to be. Note that you can change the coordinates to map any part of the texture to any part of the polygon face. In this tutorial I'm just doing a direct mapping for simplicity reasons.

```
	// Load the vertex array with data.
	vertices[0].position = D3DXVECTOR3(-1.0f, -1.0f, 0.0f);  // Bottom left.
	vertices[0].texture = D3DXVECTOR2(0.0f, 1.0f);

	vertices[1].position = D3DXVECTOR3(0.0f, 1.0f, 0.0f);  // Top middle.
	vertices[1].texture = D3DXVECTOR2(0.5f, 0.0f);

	vertices[2].position = D3DXVECTOR3(1.0f, -1.0f, 0.0f);  // Bottom right.
	vertices[2].texture = D3DXVECTOR2(1.0f, 1.0f);

}
```

LoadTexture is a new private function that will create the texture object and then initialize it with the input file name provided. This function is called during initialization.

```
bool ModelClass::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;
}
```

The ReleaseTexture function will release the texture object that was created and loaded during the LoadTexture function.

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

	return;
}
```

textureshaderclass.h

The TextureShaderClass is just an updated version of the ColorShaderClass from the previous tutorial. This class will be used to draw the 3D models using vertex and pixel shaders.

```
#ifndef _COLORSHADERCLASS_H_
#define _COLORSHADERCLASS_H_
#ifndef _TEXTURESHADERCLASS_H_
#define _TEXTURESHADERCLASS_H_

class ColorShaderClass
class TextureShaderClass
{
public:
	ColorShaderClass();
	ColorShaderClass(const ColorShaderClass&);
	~ColorShaderClass();
	TextureShaderClass();
	TextureShaderClass(const TextureShaderClass&);
	~TextureShaderClass();

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

private:
	bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX);
	bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView*);
```

There is a new private variable for the sampler state pointer. This pointer will be used to interface with the texture shader.

```
private:
	ID3D11SamplerState* m_sampleState;
};
```

textureshaderclass.cpp

```
#include "colorshaderclass.h"
#include "textureshaderclass.h"

ColorShaderClass::ColorShaderClass()
TextureShaderClass::TextureShaderClass()
{
```

The new sampler variable is set to null in the class constructor.

```
    m_sampleState = 0;
}

ColorShaderClass::ColorShaderClass(const ColorShaderClass& other)
TextureShaderClass::TextureShaderClass(const TextureShaderClass& other)
{
}

ColorShaderClass::~ColorShaderClass()
TextureShaderClass::~TextureShaderClass()
{
}

bool ColorShaderClass::Initialize(ID3D11Device* device, HWND hwnd)
bool TextureShaderClass::Initialize(ID3D11Device* device, HWND hwnd)
{
```

The new texture.vs and texture.ps HLSL files are loaded for this shader.

```
    // Initialize the vertex and pixel shaders.
    result = InitializeShader(device, hwnd, L"../Engine/color.vs", L"../Engine/color.ps");
    // Initialize the vertex and pixel shaders.
    result = InitializeShader(device, hwnd, L"../Engine/texture.vs", L"../Engine/texture.ps");
}
```

The Shutdown function calls the release of the shader variables.

```
void ColorShaderClass::Shutdown()
void TextureShaderClass::Shutdown()
{
}
```

The Render function now takes a new parameter called texture which is the pointer to the texture resource. This is then sent into the SetShaderParameters function so that the texture can be set in the shader and then used for rendering.

```
bool ColorShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
    D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix)
bool TextureShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
    D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture)
{
}
```

InitializeShader sets up the texture shader.

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

We have a new variable to hold the description of the texture sampler that will be setup in this function.

```
    D3D11_SAMPLER_DESC samplerDesc;
```

Load in the new texture vertex and pixel shaders.

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

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

The input layout has changed as we now have a texture element instead of color. The first position element stays unchanged but the SemanticName and Format of the second element have been changed to TEXCOORD and DXGI_FORMAT_R32G32_FLOAT. These two changes will now align this layout with our new VertexType in both the ModelClass definition and the typedefs in the shader files.

```
polygonLayout[1].SemanticName = "COLOR";
polygonLayout[1].SemanticName = "TEXCOORD";
polygonLayout[1].SemanticIndex = 0;
polygonLayout[1].Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
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;
```

The sampler state description is setup here and then can be passed to the pixel shader after. The most important element of the texture sampler description is **Filter**. Filter will determine how it decides which pixels will be used or combined to create the final look of the texture on the polygon face. In the example here we use D3D11_FILTER_MIN_MAG_MIP_LINEAR which is more expensive in terms of processing but gives the best visual result. It tells the sampler to use **linear interpolation for minification, magnification,** and **mip-level sampling.**

AddressU and AddressV are set to Wrap which ensures that the coordinates stay between 0.0f and 1.0f. Anything outside of that wraps around and is placed between 0.0f and 1.0f. All other settings for the sampler state description are defaults.

```
// 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;
}
```

The ShutdownShader function releases all the variables used in the TextureShaderClass.

```
~~void ColorShaderClass::ShutdownShader()~~
void TextureShaderClass::ShutdownShader()
{
```

The ShutdownShader function now releases the new sampler state that was created during initialization.

```
	// Release the sampler state.
	if(m_sampleState)
	{
		m_sampleState->Release();
		m_sampleState = 0;
	}
}
```

OutputShaderErrorMessage writes out errors to a text file if the HLSL shader could not be loaded.

```
~~void ColorShaderClass::OutputShaderErrorMessage(ID3D10Blob* errorMessage, HWND hwnd, WCHAR*
	shaderFilename)~~
void TextureShaderClass::OutputShaderErrorMessage(ID3D10Blob* errorMessage, HWND hwnd, WCHAR*
	shaderFilename)
{
}
```

SetShaderParameters function now takes in a pointer to a texture resource and then assigns it to the shader using the new texture resource pointer. Note that the texture must be set before rendering of the buffer occurs.

```
~~bool ColorShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix)~~
bool TextureShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture)
{
```

The SetShaderParameters function has been modified from the previous tutorial to include setting the texture in the pixel shader now.

```
	// Set shader texture resource in the pixel shader.
	deviceContext->PSSetShaderResources(0, 1, &texture);

	return true;
}
```

RenderShader calls the shader technique to render the polygons.

```
~~void ColorShaderClass::RenderShader(ID3D11DeviceContext* deviceContext, int indexCount)~~
void TextureShaderClass::RenderShader(ID3D11DeviceContext* deviceContext, int indexCount)
{
```

The RenderShader function has been changed to include setting the sample state in the pixel shader before rendering.

```
	// Set the sampler state in the pixel shader.
	deviceContext->PSSetSamplers(0, 1, &m_sampleState);

	// Render the triangle.
	deviceContext->DrawIndexed(indexCount, 0, 0);

	return;
}
```

graphicsclass.h

The GraphicsClass now includes the new TextureShaderClass header and the ColorShaderClass header has been removed.

```
#include "textureshaderclass.h"

class GraphicsClass
{
```

A new TextureShaderClass private object has been added.

```
private:
	TextureShaderClass* m_TextureShader;
};
```

graphicsclass.cpp

```
GraphicsClass::GraphicsClass()
{
```

The m_TextureShader variable is set to null in the constructor.

```
	m_TextureShader = 0;
}

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

The ModelClass::Initialize function now takes in the name of the texture that will be used for rendering the model.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice());
	result = m_Model->Initialize(m_D3D->GetDevice(), L"../Engine/data/seafloor.dds");
```

The new TextureShaderClass object is created and initialized.

```
	// Create the color shader object.
	m_ColorShader = new ColorShaderClass;
	if(!m_ColorShader)
	// Create the texture shader object.
	m_TextureShader = new TextureShaderClass;
	if(!m_TextureShader)
}

void GraphicsClass::Shutdown()
{
```

The TextureShaderClass object is also released in the Shutdown function.

```
	// Release the color shader object.
	if(m_ColorShader)
	{
		m_ColorShader->Shutdown();
		delete m_ColorShader;
		m_ColorShader = 0;
	}
	// Release the texture shader object.
	If (m_TextureShader)
	{
		m_TextureShader->Shutdown();
		delete m_TextureShader;
		m_TextureShader = 0;
	}
}
```

```
bool GraphicsClass::Render()
{
```

The texture shader is called now instead of the color shader to render the model. Notice it also takes the texture resource pointer from the model so the texture shader has access to the texture from the model object.

```
    // Render the model using the color shader.
    result = m_ColorShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
                                   viewMatrix, projectionMatrix);
    // Render the model using the texture shader.
    result = m_TextureShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(),
                                     worldMatrix, viewMatrix, projectionMatrix, m_Model->GetTexture());
}
```

Summary

You should now understand the basics of loading a texture, mapping it to a polygon face, and then rendering it with a shader.

Exercise

1. Create your own DDS texture (try to find one with patterns) and change the code to create two triangles that form a square. Map the entire texture (your texture) to this square so that the entire texture shows up correctly on the screen.
2. Create another square and place it further in 3D space to show the effect of the linear sampling (MIN_MAG_MIP_LINEAR) filter and the point sampling (MIN_MAG_MIP_POINT) filter. Use key numbers (1: MIN_MAG_MIP_LINEAR, 2: MIN_MAG_MIP_POINT) such that you can change the filter mode without recompiling the code.