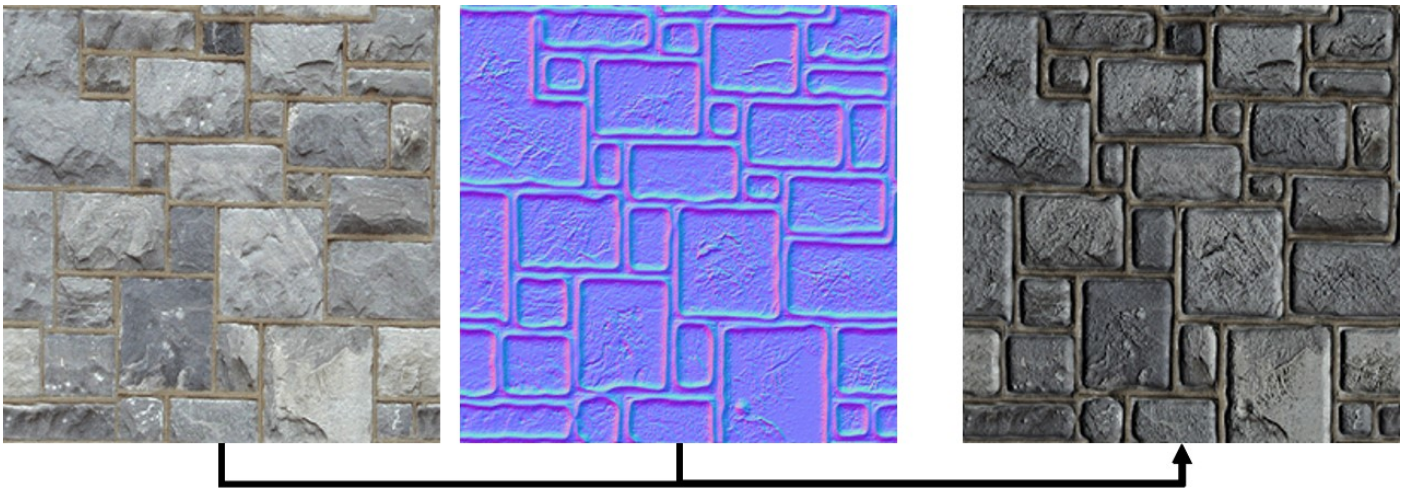
Tutorial: Bump Mapping

This tutorial will cover how to perform bump mapping in DirectX 11 using HLSL and C++. The proper terminology for the bump mapping technique is called normal mapping. It uses a special texture called a normal map which is essentially a look up table for surface normals. Each pixel in this normal map indicates the light direction for the corresponding pixel on the texture color map.

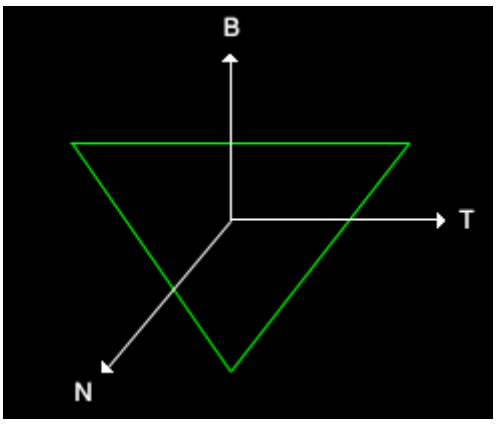
For example, take the following color (left) and normal (middle) map which produce the following bump mapped texture (right):



As you can see the effect is very realistic and the cost of producing it using bump mapping is far less expensive then rendering a high polygon surface to get the same result.

To create a normal map, you usually need someone to produce a 3D model of the surface and then use a tool to convert that 3D model into a normal map. There are also certain tools that will work with 2D textures to produce a somewhat decent normal map, but it is obviously not as accurate as the 3D model version would be.

The tools that create normal maps take the x, y, z coordinates and translate them to red, green, blue pixels with the intensity of each color indicating the angle of the normal they represent. The normal of our polygon surface is still calculated the same way as before. However, the two other normals we need to calculate require the vertex and texture coordinates for that polygon surface. These two normals are called the tangent and binormal. The diagram below shows the direction of each normal:



The normal is still pointing straight out towards the viewer. The tangent and binormal however run across the surface of the polygon with the tangent going along the x-axis and the binormal going along the y-axis. These two normals then directly translate to the tu and tv texture coordinates of the normal map with the texture U coordinate mapping to the tangent and the texture V coordinate mapping to the binormal.

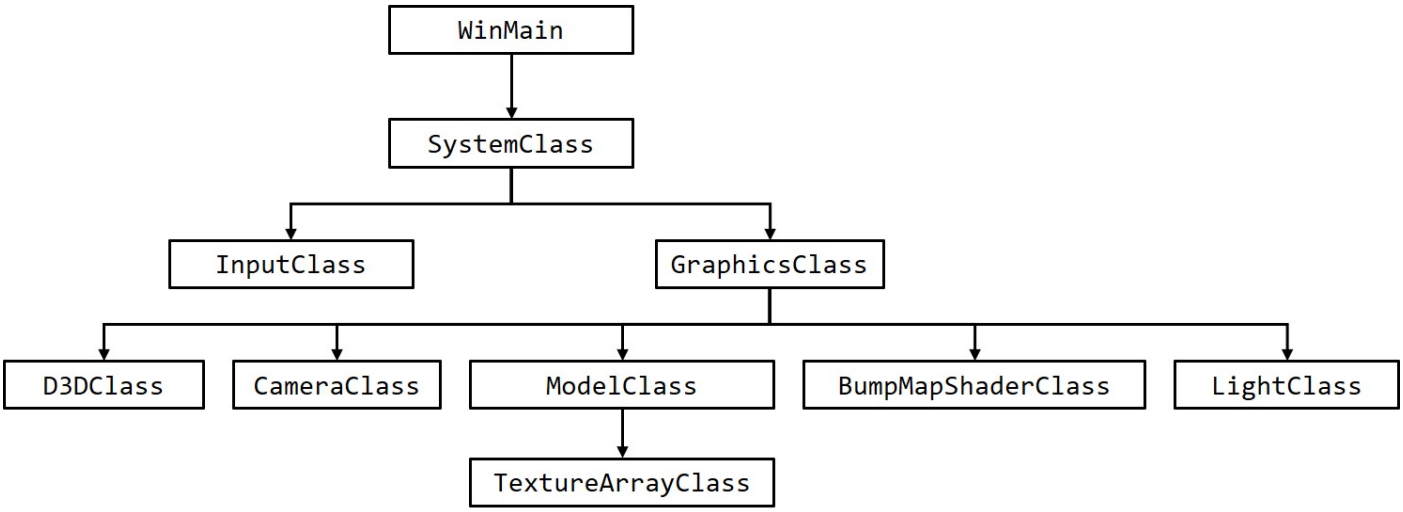
We will need to do some precalculation to determine the binormal and tangent vector using the normal and texture coordinates. Also note that you should never do this inside the shader as it is fairly expensive with all the floating point math involved, I instead use a function in my C++ code that you will see to do this during the model loading. Also if you are looking to use this effect on a large number of high polygon models it may be best to precalculate these

different normals and store them in your model format. Once we have precalculated the tangent and binormal we can use this equation to determine the bump normal at any pixel using the normal map:

bumpNormal = (bumpMap.x * input.tangent) + (bumpMap.y * input.binormal) + (bumpMap.z * input.normal);

Once we have the normal for that pixel we can then calculate against the light direction and multiply by the color value of the pixel from the color texture to get the final result.

Framework



Bumpmap.vs

```
////////////////////////////////////////////////////////////////////////////////
// Filename: bumpmap.vs
////////////////////////////////////////////////////////////////////////////////

/////////////
// GLOBALS //
/////////////
cbuffer MatrixBuffer
{
    matrix worldMatrix;
    matrix viewMatrix;
    matrix projectionMatrix;
};
```

Both the VertexInputType and PixelInputType now have a tangent and binormal vector for bump map calculations.

```
///////////////
// TYPEDEFS //
///////////////
struct VertexInputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 tangent : TANGENT;
    float3 binormal : BINORMAL;
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 tangent : TANGENT;
    float3 binormal : BINORMAL;
```

```
};

////////////////////////////////////////////////////////////////////////////////
// Vertex Shader
////////////////////////////////////////////////////////////////////////////////
PixelInputType BumpMapVertexShader(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);
```

Both the input tangent and binormal are calculated against the world matrix and then normalized the same as the input normal vector.

```
    // 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);

    return output;
}
```

Bumpmap.ps

```
////////////////////////////////////////////////////////////////////////////////
// Filename: bumpmap.ps
////////////////////////////////////////////////////////////////////////////////

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

The bump map shader requires two textures. The first texture in the array is the color texture. The second texture is the normal map.

```
Texture2D shaderTextures[2];
SamplerState SampleType;
```

Just like most light shaders the direction and color of the light is required for lighting calculations.

```
cbuffer LightBuffer
{
    float4 diffuseColor;
    float3 lightDirection;
};

//////////////
// TYPEDEFS //
//////////////
```

```
struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 tangent : TANGENT;
    float3 binormal : BINORMAL;
};
```

The pixel shader works as we described above with a couple additional lines of code. First we sample the pixel from the color texture and the normal map. We then multiply the normal map value by two and then subtract one to move it into the -1.0 to +1.0 float range. We have to do this because the sampled value that is presented to us in the 0.0 to +1.0 texture range which only covers half the range we need for bump map normal calculations. After that we then calculate the bump normal which uses the equation we described earlier. This bump normal is normalized and then used to determine the light intensity at this pixel by doing a dot product with the light direction. Once we have the light intensity at this pixel the bump mapping is now done. We use the light intensity with the light color and texture color to get the final pixel color.

```
////////////////////////////////////////////////////////////////////////////////
// Pixel Shader
////////////////////////////////////////////////////////////////////////////////
float4 BumpMapPixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor;
    float4 bumpMap;
    float3 bumpNormal;
    float3 lightDir;
    float lightIntensity;
    float4 color;

    // Sample the texture pixel at this location.
    textureColor = shaderTextures[0].Sample(SampleType, input.tex);

    // Sample the pixel in the bump map.
    bumpMap = shaderTextures[1].Sample(SampleType, input.tex);

    // Expand the range of the normal value from (0, +1) to (-1, +1).
    bumpMap = (bumpMap * 2.0f) - 1.0f;

    // Calculate the normal from the data in the bump map.
    bumpNormal = (bumpMap.x * input.tangent) + (bumpMap.y * input.binormal) + (bumpMap.z * input.normal);

    // Normalize the resulting bump normal.
    bumpNormal = normalize(bumpNormal);

    // Invert the light direction for calculations.
    lightDir = -lightDirection;

    // Calculate the amount of light on this pixel based on the bump map normal value.
    lightIntensity = saturate(dot(bumpNormal, lightDir));

    // Determine the final diffuse color based on the diffuse color and the amount of light intensity.
    color = saturate(diffuseColor * lightIntensity);

    // Combine the final bump light color with the texture color.
    color = color * textureColor;

    return color;
}
```

Bumpmapshaderclass.h

The BumpMapShaderClass is just a modified version of the shader classes from the previous tutorials.

```
////////////////////////////////////////////////////////////////////////////////
```

```
// Filename: bumpmapshaderclass.h
////////////////////////////////////////////////////////////////////////////
#ifndef _BUMPMAPSHADERCLASS_H_
#define _BUMPMAPSHADERCLASS_H_

//////////////
// INCLUDES //
//////////////
#include <d3d11.h>
#include <d3dx10math.h>
#include <d3dx11async.h>
#include <fstream>
using namespace std;

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

	struct LightBufferType
	{
		D3DXVECTOR4 diffuseColor;
		D3DXVECTOR3 lightDirection;
		float padding;
	};

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

	bool Initialize(ID3D11Device*, HWND);
	void Shutdown();
	bool Render(ID3D11DeviceContext*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView**, D3DXVECTOR3, D3DXVECTOR4);

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

	bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView**, D3DXVECTOR3, D3DXVECTOR4);
	void RenderShader(ID3D11DeviceContext*, int);

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

The bump map shader will require a constant buffer to interface with the light direction and light color.

```
	ID3D11Buffer* m_lightBuffer;
};

#endif
```

Bumpmapshaderclass.cpp

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

The class constructor initializes the pointers to null.

```
BumpMapShaderClass::BumpMapShaderClass()
{
	m_vertexShader = 0;
	m_pixelShader = 0;
	m_layout = 0;
	m_matrixBuffer = 0;
	m_sampleState = 0;
	m_lightBuffer = 0;
}

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

BumpMapShaderClass::~BumpMapShaderClass()
{
}
```

The Initialize function will call the shader to load the bump map HLSL files.

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

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

	return true;
}
```

Shutdown releases the shader effect.

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

	return;
}
```

The Render function sets the shader parameters first and then renders the model using the bump map shader.

```
bool BumpMapShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount,
	D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix,
	ID3D11ShaderResourceView** textureArray, D3DXVECTOR3 lightDirection, D3DXVECTOR4 diffuseColor)
{
	bool result;

	// Set the shader parameters that it will use for rendering.
	result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, textureArray,
		lightDirection, diffuseColor);
```

```
	if(!result)
	{
		return false;
	}

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

	return true;
}
```

InitializeShader sets up the bump map shader.

```
bool BumpMapShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR*
	psFilename)
{
	HRESULT result;
	ID3D10Blob* errorMessage;
	ID3D10Blob* vertexShaderBuffer;
	ID3D10Blob* pixelShaderBuffer;
```

The polygon layout is now set to five elements to accommodate the tangent and binormal.

```
	D3D11_INPUT_ELEMENT_DESC polygonLayout[5];
	unsigned int numElements;
	D3D11_BUFFER_DESC matrixBufferDesc;
	D3D11_SAMPLER_DESC samplerDesc;
	D3D11_BUFFER_DESC lightBufferDesc;

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

The bump map vertex shader is loaded here.

```
	// Compile the vertex shader code.
	result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "BumpMapVertexShader", "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 bump map pixel shader is loaded here.

```
	// Compile the pixel shader code.
	result = D3DX11CompileFromFile(psFilename, NULL, NULL, "BumpMapPixelShader", "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;

	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;
```

The layout now includes a tangent and binormal element which are setup the same as the normal element with the exception of the semantic name.

```
	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;

	// 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;
	}
```

The light constant buffer is setup here.

```
	// 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 vertex shader constant buffer from within this class.
	result = device->CreateBuffer(&lightBufferDesc, NULL, &m_lightBuffer);
	if(FAILED(result))
	{
		return false;
	}

	return true;
}
```

The ShutdownShader function releases all the pointers that were setup in the InitializeShader function.

```
void BumpMapShaderClass::ShutdownShader()
{
	// Release the light constant buffer.
	if(m_lightBuffer)
	{
		m_lightBuffer->Release();
		m_lightBuffer = 0;
	}

	// 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;
}
```

OutputShaderErrorMessage writes out errors to a text file if the HLSL shader file won't compile properly.

```
void BumpMapShaderClass::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 SetShaderParameters function sets the shader parameters before rendering occurs.

```
bool BumpMapShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext,
        D3DXMATRIX worldMatrix, D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix,
        ID3D11ShaderResourceView** textureArray, D3DXVECTOR3 lightDirection, D3DXVECTOR4 diffuseColor)
{
        HRESULT result;
        D3D11_MAPPED_SUBRESOURCE mappedResource;
        MatrixBufferType* dataPtr;
        unsigned int bufferNumber;
        LightBufferType* dataPtr2;

        // 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 texture array is set here, it contains two textures. The first texture is the color texture and the second texture is the normal map.

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

The light buffer in the pixel shader is then set with the diffuse light color and light direction.

```
	// 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->diffuseColor = diffuseColor;
	dataPtr2->lightDirection = lightDirection;

	// 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);

	return true;
}
```

RenderShader draws the model using the bump map shader.

```
void BumpMapShaderClass::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;
}
```

Modelclass.h

```
////////////////////////////////////////////////////////////////////////////////
// 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:
```

The VertexType structure has been changed to now have a tangent and binormal vector.

```
	struct VertexType
	{
		D3DXVECTOR3 position;
		D3DXVECTOR2 texture;
		D3DXVECTOR3 normal;
		D3DXVECTOR3 tangent;
		D3DXVECTOR3 binormal;
	};
```

The ModelType structure has also been changed to have a tangent and binormal vector.

```
	struct ModelType
	{
		float x, y, z;
		float tu, tv;
		float nx, ny, nz;
		float tx, ty, tz;
		float bx, by, bz;
	};
```

The following two structures will be used for calculating the tangent and binormal.

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

	struct VectorType
	{
		float x, y, z;
	};
```

```
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();
```

We have three new functions for calculating the tangent and binormal vectors for the model.

```
        void CalculateModelVectors();
        void CalculateTangentBinormal(TempVertexType, TempVertexType, TempVertexType, VectorType&,
                VectorType&);
        void CalculateNormal(VectorType, VectorType, VectorType&);

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

#endif
```

Modelclass.cpp

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

The Initialize function now takes in two texture filenames. The first texture filename is for the color texture. The second texture filename is for the normal map that will be used to create the bump effect.

```
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;
        }
```

After the model data has been loaded we now call the new CalculateModelVectors function to calculate the tangent and binormal. It also recalculates the normal vector.

```
        // Calculate the normal, tangent, and binormal vectors for the model.
```

```
	CalculateModelVectors();

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

The two textures for the model are loaded here. The first is the color texture and the second is the normal map.

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

	return true;
}

bool ModelClass::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;
	}
```

The InitializeBuffers function has changed at this point where the vertex array is loaded with data from the ModelType array. The ModelType array now has tangent and binormal values for the model so they need to be copied into the vertex array which will then be copied into the vertex buffer.

```
	// 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);
		vertices[i].texture = D3DXVECTOR2(m_model[i].tu, m_model[i].tv);
		vertices[i].normal = D3DXVECTOR3(m_model[i].nx, m_model[i].ny, m_model[i].nz);
		vertices[i].tangent = D3DXVECTOR3(m_model[i].tx, m_model[i].ty, m_model[i].tz);
		vertices[i].binormal = D3DXVECTOR3(m_model[i].bx, m_model[i].by, m_model[i].bz);

		indices[i] = i;
	}

	// 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 vertex and index buffers have been created and loaded.
	delete [] vertices;
	vertices = 0;

	delete [] indices;
	indices = 0;

	return true;
}
```

LoadTextures now creates a vertex array and then loads the color texture and normal map into the two element texture array.

```
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;
```

```
}
```

CalculateModelVectors generates the tangent and binormal for the model as well as a recalculated normal vector. To start it calculates how many faces (triangles) are in the model. Then for each of those triangles it gets the three vertices and uses that to calculate the tangent, binormal, and normal. After calculating those three normal vectors it then saves them back into the model structure.

```
void ModelClass::CalculateModelVectors()
{
	int faceCount, i, index;
	TempVertexType vertex1, vertex2, vertex3;
	VectorType tangent, binormal, normal;

	// Calculate the number of faces in the model.
	faceCount = m_vertexCount / 3;

	// Initialize the index to the model data.
	index = 0;

	// Go through all the faces and calculate the the tangent, binormal, and normal vectors.
	for(i=0; i<faceCount; i++)
	{
		// Get the three vertices for this face from the model.
		vertex1.x = m_model[index].x;
		vertex1.y = m_model[index].y;
		vertex1.z = m_model[index].z;
		vertex1.tu = m_model[index].tu;
		vertex1.tv = m_model[index].tv;
		vertex1.nx = m_model[index].nx;
		vertex1.ny = m_model[index].ny;
		vertex1.nz = m_model[index].nz;
		index++;

		vertex2.x = m_model[index].x;
		vertex2.y = m_model[index].y;
		vertex2.z = m_model[index].z;
		vertex2.tu = m_model[index].tu;
		vertex2.tv = m_model[index].tv;
		vertex2.nx = m_model[index].nx;
		vertex2.ny = m_model[index].ny;
		vertex2.nz = m_model[index].nz;
		index++;

		vertex3.x = m_model[index].x;
		vertex3.y = m_model[index].y;
		vertex3.z = m_model[index].z;
		vertex3.tu = m_model[index].tu;
		vertex3.tv = m_model[index].tv;
		vertex3.nx = m_model[index].nx;
		vertex3.ny = m_model[index].ny;
		vertex3.nz = m_model[index].nz;
		index++;

		// Calculate the tangent and binormal of that face.
		CalculateTangentBinormal(vertex1, vertex2, vertex3, tangent, binormal);

		// Calculate the new normal using the tangent and binormal.
		CalculateNormal(tangent, binormal, normal);

		// Store the normal, tangent, and binormal for this face back in the model structure.
		m_model[index-1].nx = normal.x;
		m_model[index-1].ny = normal.y;
		m_model[index-1].nz = normal.z;
		m_model[index-1].tx = tangent.x;
		m_model[index-1].ty = tangent.y;
		m_model[index-1].tz = tangent.z;
```

```
                m_model[index-1].bx = binormal.x;
                m_model[index-1].by = binormal.y;
                m_model[index-1].bz = binormal.z;

                m_model[index-2].nx = normal.x;
                m_model[index-2].ny = normal.y;
                m_model[index-2].nz = normal.z;
                m_model[index-2].tx = tangent.x;
                m_model[index-2].ty = tangent.y;
                m_model[index-2].tz = tangent.z;
                m_model[index-2].bx = binormal.x;
                m_model[index-2].by = binormal.y;
                m_model[index-2].bz = binormal.z;

                m_model[index-3].nx = normal.x;
                m_model[index-3].ny = normal.y;
                m_model[index-3].nz = normal.z;
                m_model[index-3].tx = tangent.x;
                m_model[index-3].ty = tangent.y;
                m_model[index-3].tz = tangent.z;
                m_model[index-3].bx = binormal.x;
                m_model[index-3].by = binormal.y;
                m_model[index-3].bz = binormal.z;
        }

        return;
}
```

The CalculateTangentBinormal function takes in three vertices and then calculates and returns the tangent and binormal of those three vertices.

```
void ModelClass::CalculateTangentBinormal(TempVertexType vertex1, TempVertexType vertex2,
        TempVertexType vertex3, VectorType& tangent, VectorType& binormal)
{
        float vector1[3], vector2[3];
        float tuVector[2], tvVector[2];
        float den;
        float length;

        // Calculate the two vectors for this face.
        vector1[0] = vertex2.x - vertex1.x;
        vector1[1] = vertex2.y - vertex1.y;
        vector1[2] = vertex2.z - vertex1.z;

        vector2[0] = vertex3.x - vertex1.x;
        vector2[1] = vertex3.y - vertex1.y;
        vector2[2] = vertex3.z - vertex1.z;

        // Calculate the tu and tv texture space vectors.
        tuVector[0] = vertex2.tu - vertex1.tu;
        tvVector[0] = vertex2.tv - vertex1.tv;

        tuVector[1] = vertex3.tu - vertex1.tu;
        tvVector[1] = vertex3.tv - vertex1.tv;

        // Calculate the denominator of the tangent/binormal equation.
        den = 1.0f / (tuVector[0] * tvVector[1] - tuVector[1] * tvVector[0]);

        // Calculate the cross products and multiply by the coefficient to get the tangent and binormal.
        tangent.x = (tvVector[1] * vector1[0] - tvVector[0] * vector2[0]) * den;
        tangent.y = (tvVector[1] * vector1[1] - tvVector[0] * vector2[1]) * den;
        tangent.z = (tvVector[1] * vector1[2] - tvVector[0] * vector2[2]) * den;

        binormal.x = (tuVector[0] * vector2[0] - tuVector[1] * vector1[0]) * den;
        binormal.y = (tuVector[0] * vector2[1] - tuVector[1] * vector1[1]) * den;
        binormal.z = (tuVector[0] * vector2[2] - tuVector[1] * vector1[2]) * den;
```

```
	// Calculate the length of this normal.
	length = sqrt((tangent.x * tangent.x) + (tangent.y * tangent.y) + (tangent.z * tangent.z));

	// Normalize the normal and then store it
	tangent.x = tangent.x / length;
	tangent.y = tangent.y / length;
	tangent.z = tangent.z / length;

	// Calculate the length of this normal.
	length = sqrt((binormal.x * binormal.x) + (binormal.y * binormal.y) + (binormal.z * binormal.z));

	// Normalize the normal and then store it
	binormal.x = binormal.x / length;
	binormal.y = binormal.y / length;
	binormal.z = binormal.z / length;

	return;
}
```

The CalculateNormal function takes in the tangent and binormal and then does a cross product to give back the normal vector.

```
void ModelClass::CalculateNormal(VectorType tangent, VectorType binormal, VectorType& normal)
{
	float length;

	// Calculate the cross product of the tangent and binormal which will give the normal vector.
	normal.x = (tangent.y * binormal.z) - (tangent.z * binormal.y);
	normal.y = (tangent.z * binormal.x) - (tangent.x * binormal.z);
	normal.z = (tangent.x * binormal.y) - (tangent.y * binormal.x);

	// Calculate the length of the normal.
	length = sqrt((normal.x * normal.x) + (normal.y * normal.y) + (normal.z * normal.z));

	// Normalize the normal.
	normal.x = normal.x / length;
	normal.y = normal.y / length;
	normal.z = normal.z / length;

	return;
}
```

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 BumpMapShaderClass header file is included here in the GraphicsClass header.

```
#include "bumpmapshaderclass.h"
#include "lightclass.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;
```

The new BumpMapShaderClass object is created here.

```
	BumpMapShaderClass* m_BumpMapShader;
	LightClass* m_Light;
};

#endif
```

Graphicsclass.cpp

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

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

We initialize the BumpMapShaderClass object to null in the class constructor.

```
	m_BumpMapShader = 0;
	m_Light = 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 the cube model, the stone01.dds color texture, and the bump01.dds normal map.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), "../Engine/data/cube.txt", L"../Engine/data/stone01.dds",
		L"../Engine/data/bump01.dds");
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB_OK);
		return false;
	}
```

Here we create and initialize the BumpMapShaderClass object.

```
	// Create the bump map shader object.
	m_BumpMapShader = new BumpMapShaderClass;
	if(!m_BumpMapShader)
	{
		return false;
	}

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

	// Create the light object.
	m_Light = new LightClass;
	if(!m_Light)
	{
		return false;
	}
```

The light color is set to white and the light direction is set down the positive Z axis.

```
	// Initialize the light object.
	m_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);
	m_Light->SetDirection(0.0f, 0.0f, 1.0f);

	return true;
}

void GraphicsClass::Shutdown()
{
	// Release the light object.
	if(m_Light)
	{
		delete m_Light;
		m_Light = 0;
	}
```

The new BumpMapShaderClass is released here in the Shutdown function.

```
	// Release the bump map shader object.
	if(m_BumpMapShader)
	{
		m_BumpMapShader->Shutdown();
		delete m_BumpMapShader;
		m_BumpMapShader = 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;
	static float rotation = 0.0f;

	// 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);
```

Rotate the cube model each frame to show the effect.

```
	// Update the rotation variable each frame.
	rotation += (float)D3DX_PI * 0.0025f;
	if(rotation > 360.0f)
	{
		rotation -= 360.0f;
	}

	// Rotate the world matrix by the rotation value.
	D3DXMatrixRotationY(&worldMatrix, rotation);

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

Render the model using the bump map shader.

```
	// Render the model using the bump map shader.
	m_BumpMapShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
		viewMatrix, projectionMatrix, m_Model->GetTextureArray(), m_Light->GetDirection(),
		m_Light->GetDiffuseColor());

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

	return true;
}
```

Summary

With the bump map shader you can create very detailed scenes that look 3D with just two 2D textures.

To Do Exercises

1. Recompile and run the program. You should see a bump mapped rotating cube. Press escape to quit.

2. Change the bump map effect from 2.5 to something smaller (like 1.0) and larger (like 5.0) in the shader to see the change in the bump depth.

3. Comment out the color = color * textureColor; line in the pixel shader to see just the bump lighting effect.

4. Move the camera and light position around to see the effect from different angles.

[Original script: www.rastertek.com]