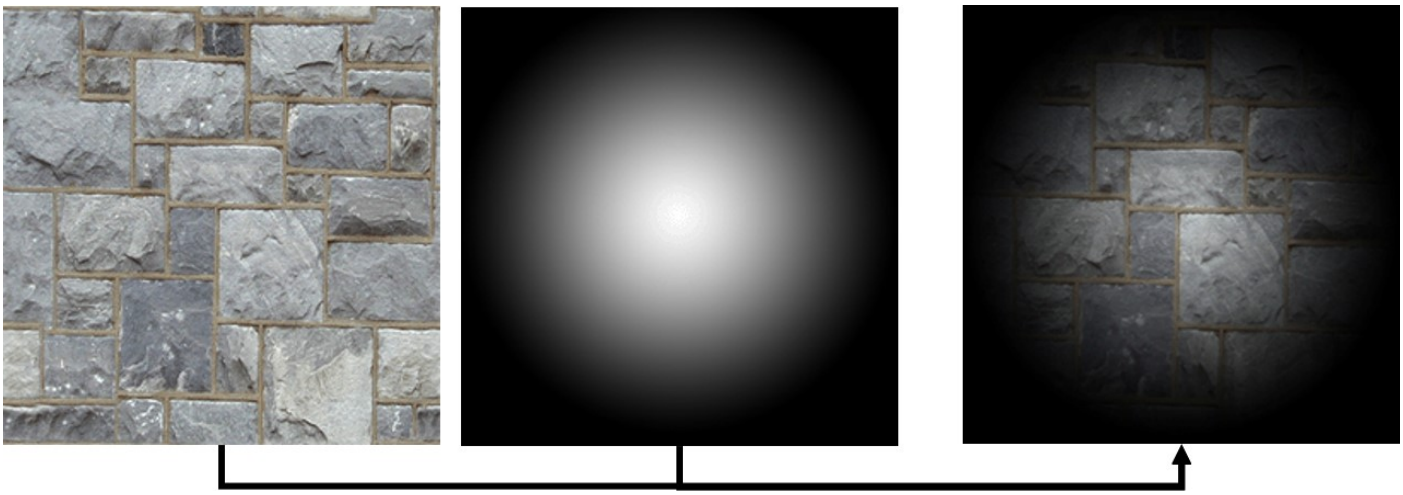
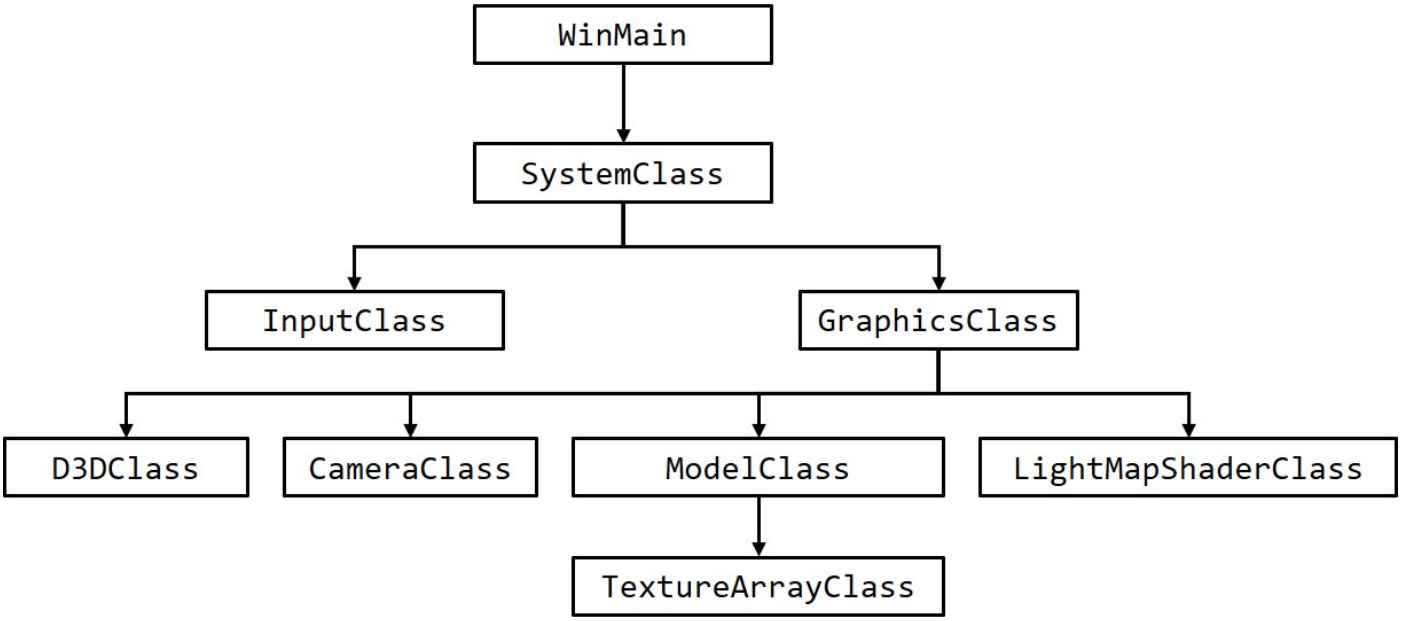
Tutorial: Light Maps

Light mapping in DirectX 11 is the process of using a secondary texture or data file to create a fast look up table to create unique lighting effects that require very little processing. Because a secondary source is used as the basis for our lighting, any other light calculations can be removed from the application, which can gain an incredible speed.

With light mapping we require two textures. The first texture is the base color texture. The second texture we need is the light map. Usually this is just a black and white texture with white representing the intensity of the light at each pixel. I created a spotlight style light map that we will use in this tutorial. Once, we have our color texture and our light map we can combine them in the pixel shader to produce the light mapped texture. The shader is very simple as we just multiply the two pixels together which will produce the following output:



Framework



Lightmap.vs

The light map vertex shader is the same as the multitexture vertex shader from the previous tutorial. The only thing that has changed is the name.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: lightmap.vs
////////////////////////////////////////////////////////////////////////////////

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

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

//////////////
// TYPEDEFS //
//////////////
struct VertexInputType
{
	float4 position : POSITION;
	float2 tex : TEXCOORD0;
};

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


////////////////////////////////////////////////////////////////////////////////
// Vertex Shader
////////////////////////////////////////////////////////////////////////////////
PixelInputType LightMapVertexShader(VertexInputType input)
{
	PixelInputType output;


	// Change the position vector to be 4 units for proper matrix calculations.
	input.position.w = 1.0f;

	// Calculate the position of the vertex against the world, view, and projection matrices.
	output.position = mul(input.position, worldMatrix);
	output.position = mul(output.position, viewMatrix);
	output.position = mul(output.position, projectionMatrix);

	// Store the texture coordinates for the pixel shader.
	output.tex = input.tex;

	return output;
}
```

Lightmap.ps

```
////////////////////////////////////////////////////////////////////////////////
// Filename: lightmap.ps
////////////////////////////////////////////////////////////////////////////////


/////////////
// GLOBALS //
/////////////
Texture2D shaderTextures[2];
SamplerState SampleType;


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

```
};
```

The light map pixel shader is very simple. It multiplies the color texture pixel and the light map texture value to get the desired output. This is not much different from just a regular multitexture blend other than there is no need to correct for gamma.

```
////////////////////////////////////////////////////////////////////////////////
// Pixel Shader
////////////////////////////////////////////////////////////////////////////////
float4 LightMapPixelShader(PixelInputType input) : SV_TARGET
{
    float4 color;
    float4 lightColor;
    float4 finalColor;


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

    // Get the pixel color from the light map.
    lightColor = shaderTextures[1].Sample(SampleType, input.tex);

    // Blend the two pixels together.
    finalColor = color * lightColor;

    return finalColor;
}
```

Lightmapshaderclass.h

The LightMapShaderClass is just the MultiTextureShaderClass from the previous that has now been updated for light mapping.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: lightmapshaderclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _LIGHTMAPSHADERCLASS_H_
#define _LIGHTMAPSHADERCLASS_H_

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

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

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

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

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

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

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

#endif
```

Lightmapshaderclass.cpp

I will just go over the changes since the previous tutorial. Other than the name of the functions there are only a couple changes.

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

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

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

LightMapShaderClass::~LightMapShaderClass()
{
}

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

We now load in the lightmap.vs and lightmap.ps HLSL shader files.

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

```
	return true;
}

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

	return;
}

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


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

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

	return true;
}

bool LightMapShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR*
	psFilename)
{
	HRESULT result;
	ID3D10Blob* errorMessage;
	ID3D10Blob* vertexShaderBuffer;
	ID3D10Blob* pixelShaderBuffer;
	D3D11_INPUT_ELEMENT_DESC polygonLayout[2];
	unsigned int numElements;
	D3D11_BUFFER_DESC matrixBufferDesc;
	D3D11_SAMPLER_DESC samplerDesc;

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

The light map vertex shader is loaded here.

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

```
	// Compile the pixel shader code.
	result = D3DX11CompileFromFile(psFilename, NULL, NULL, "LightMapPixelShader", "ps_5_0",
		D3D10_SHADER_ENABLE_STRICTNESS, 0, NULL, &pixelShaderBuffer, &errorMessage,
		NULL);
	if(FAILED(result))
	{
		// If the shader failed to compile it should have writen something to the error message.
		if(errorMessage)
		{
			OutputShaderErrorMessage(errorMessage, hwnd, psFilename);
		}
		// If there was    nothing in the error message then it simply could not find the file itself.
		else
		{
			MessageBox(hwnd, psFilename, L"Missing Shader File", MB_OK);
		}

		return false;
	}

	// Create the vertex shader from the buffer.
	result = device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(),
		vertexShaderBuffer->GetBufferSize(), NULL, &m_vertexShader);
	if(FAILED(result))
	{
		return false;
	}

	// Create the vertex shader from the buffer.
	result = device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(),
		pixelShaderBuffer->GetBufferSize(), NULL, &m_pixelShader);
	if(FAILED(result))
	{
		return false;
	}

	// Create the vertex input layout description.
	// This setup needs to match the VertexType stucture in the ModelClass and in the shader.
	polygonLayout[0].SemanticName = "POSITION";
	polygonLayout[0].SemanticIndex = 0;
	polygonLayout[0].Format = DXGI_FORMAT_R32G32B32_FLOAT;
	polygonLayout[0].InputSlot = 0;
	polygonLayout[0].AlignedByteOffset = 0;
	polygonLayout[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
	polygonLayout[0].InstanceDataStepRate = 0;

	polygonLayout[1].SemanticName = "TEXCOORD";
	polygonLayout[1].SemanticIndex = 0;
	polygonLayout[1].Format = DXGI_FORMAT_R32G32_FLOAT;
	polygonLayout[1].InputSlot = 0;
	polygonLayout[1].AlignedByteOffset = D3D11_APPEND_ALIGNED_ELEMENT;
	polygonLayout[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
	polygonLayout[1].InstanceDataStepRate = 0;

	// Get a count of the elements in the layout.
	numElements = sizeof(polygonLayout) / sizeof(polygonLayout[0]);

	// Create the vertex input layout.
	result = device->CreateInputLayout(polygonLayout, numElements, vertexShaderBuffer->GetBufferPointer(),
		vertexShaderBuffer->GetBufferSize(), &m_layout);
```

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

	// Release the vertex shader buffer and pixel shader buffer since they are no longer needed.
	vertexShaderBuffer->Release();
	vertexShaderBuffer = 0;

	pixelShaderBuffer->Release();
	pixelShaderBuffer = 0;

	// Setup the description of the matrix dynamic constant buffer that is in the vertex shader.
	matrixBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
	matrixBufferDesc.ByteWidth = sizeof(MatrixBufferType);
	matrixBufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
	matrixBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
	matrixBufferDesc.MiscFlags = 0;
	matrixBufferDesc.StructureByteStride = 0;

	// Create the matrix constant buffer pointer so we can access the vertex shader constant buffer from within this class.
	result = device->CreateBuffer(&matrixBufferDesc, NULL, &m_matrixBuffer);
	if(FAILED(result))
	{
		return false;
	}

	// Create a texture sampler state description.
	samplerDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
	samplerDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
	samplerDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
	samplerDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
	samplerDesc.MipLODBias = 0.0f;
	samplerDesc.MaxAnisotropy = 1;
	samplerDesc.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
	samplerDesc.BorderColor[0] = 0;
	samplerDesc.BorderColor[1] = 0;
	samplerDesc.BorderColor[2] = 0;
	samplerDesc.BorderColor[3] = 0;
	samplerDesc.MinLOD = 0;
	samplerDesc.MaxLOD = D3D11_FLOAT32_MAX;

	// Create the texture sampler state.
	result = device->CreateSamplerState(&samplerDesc, &m_sampleState);
	if(FAILED(result))
	{
		return false;
	}

	return true;
}

void LightMapShaderClass::ShutdownShader()
{
	// Release the sampler state.
	if(m_sampleState)
	{
		m_sampleState->Release();
		m_sampleState = 0;
	}

	// Release the matrix constant buffer.
	if(m_matrixBuffer)
	{
		m_matrixBuffer->Release();
		m_matrixBuffer = 0;
	}
```

```
	// Release the layout.
	if(m_layout)
	{
		m_layout->Release();
		m_layout = 0;
	}

	// Release the pixel shader.
	if(m_pixelShader)
	{
		m_pixelShader->Release();
		m_pixelShader = 0;
	}

	// Release the vertex shader.
	if(m_vertexShader)
	{
		m_vertexShader->Release();
		m_vertexShader = 0;
	}

	return;
}

void LightMapShaderClass::OutputShaderErrorMessage(ID3D10Blob* errorMessage, HWND hwnd, WCHAR*
	shaderFilename)
{
	char* compileErrors;
	unsigned long bufferSize, i;
	ofstream fout;


	// Get a pointer to the error message text buffer.
	compileErrors = (char*)(errorMessage->GetBufferPointer());

	// Get the length of the message.
	bufferSize = errorMessage->GetBufferSize();

	// Open a file to write the error message to.
	fout.open("shader-error.txt");

	// Write out the error message.
	for(i=0; i<bufferSize; i++)
	{
		fout << compileErrors[i];
	}

	// Close the file.
	fout.close();

	// Release the error message.
	errorMessage->Release();
	errorMessage = 0;

	// Pop a message up on the screen to notify the user to check the text file for compile errors.
	MessageBox(hwnd, L"Error compiling shader.    Check shader-error.txt for message.", shaderFilename,
	MB_OK);

	return;
}

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

```
	HRESULT result;
	D3D11_MAPPED_SUBRESOURCE mappedResource;
	MatrixBufferType* dataPtr;
	unsigned int bufferNumber;

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

	// Lock the matrix constant buffer so it can be written to.
	result = deviceContext->Map(m_matrixBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
	if(FAILED(result))
	{
		return false;
	}

	// Get a pointer to the data in the constant buffer.
	dataPtr = (MatrixBufferType*)mappedResource.pData;

	// Copy the matrices into the constant buffer.
	dataPtr->world = worldMatrix;
	dataPtr->view = viewMatrix;
	dataPtr->projection = projectionMatrix;

	// Unlock the matrix constant buffer.
	deviceContext->Unmap(m_matrixBuffer, 0);

	// Set the position of the matrix constant buffer in the vertex shader.
	bufferNumber = 0;

	// Now set the matrix constant buffer in the vertex shader with the updated values.
	deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m_matrixBuffer);
```

The texture array is set the same as the previous tutorial. However instead of two color textures there is now just one color texture and the second texture in the array is a light map.

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

	return true;
}

void LightMapShaderClass::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;
}
```

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 header for the LightMapShaderClass is now included in the GraphicsClass header file.

```
#include "lightmapshaderclass.h"

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

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

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

We have a new variable for the LightMapShaderClass object.

```
	LightMapShaderClass* m_LightMapShader;
};

#endif
```

Graphicsclass.cpp

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

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

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

The new LightMapShaderClass object is initialized in the class constructor.

```
	m_LightMapShader = 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 takes as input the new light01.dds light map texture for the light map shading on this model.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), "../Engine/data/square.txt",
		L"../Engine/data/stone01.dds", L"../Engine/data/light01.dds");
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB_OK);
		return false;
	}
```

The new LightMapShaderClass object is created and initialized here.

```
	// Create the light map shader object.
	m_LightMapShader = new LightMapShaderClass;
	if(!m_LightMapShader)
	{
		return false;
	}
```

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

	return true;
}


void GraphicsClass::Shutdown()
{
```

The LightMapShaderClass object is released here in the Shutdown function.

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

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

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

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

	return;
}


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


	// Clear the buffers to begin the scene.
	m_D3D->BeginScene(0.0f, 0.0f, 0.0f, 1.0f);

	// Generate the view matrix based on the camera's position.
	m_Camera->Render();

	// Get the world, view, projection, and ortho matrices from the camera and D3D objects.
	m_D3D->GetWorldMatrix(worldMatrix);
```

```
    m_Camera->GetViewMatrix(viewMatrix);
    m_D3D->GetProjectionMatrix(projectionMatrix);
    m_D3D->GetOrthoMatrix(orthoMatrix);

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

The model is rendered here using the light map shader.

```
    // Render the model using the light map shader.
    m_LightMapShader->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

This tutorial isn't much different than the previous blending tutorial but produces a very useful effect that can be very efficient in terms of processing speed.

To Do Exercises

1. Recompile the code and ensure you get a light mapped texture on the screen.

2. Create some of your own light maps and try them out.

3. Multiply the final output pixel in the pixel shader by 2.0. Notice you can create stronger and softer lighting effects by doing this.

[Original script: www.rastertek.com]