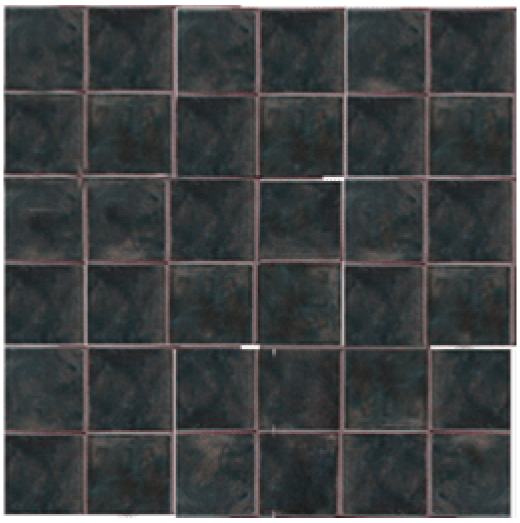
Tutorial: Specular Mapping

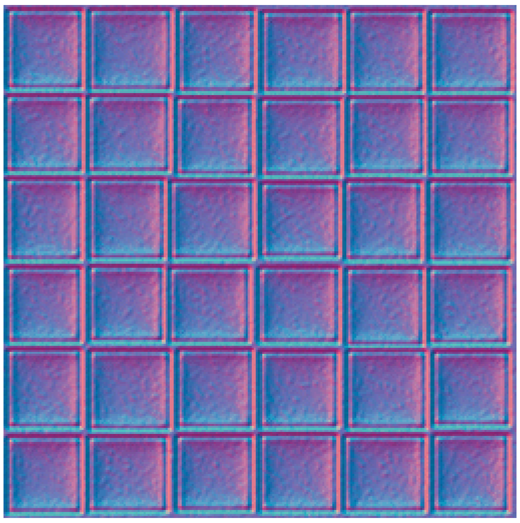
This tutorial will cover how to implement specular mapping in DirectX 11 using HLSL as well as how to combine it with bump mapping. The code in this tutorial is a combine of the code in the bump map tutorial and the specular lighting tutorial.

Specular mapping is the process of using a texture or alpha layer of a texture as a look up table for per-pixel specular lighting intensity. This process is identical to how light mapping works except that it is used for specular highlights instead.

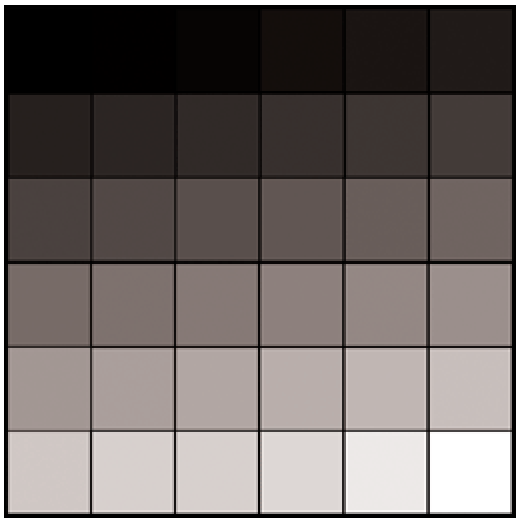
For example we may use a color texture as follows:



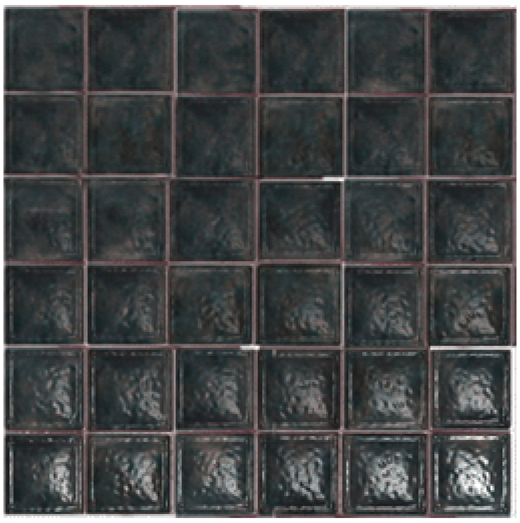
And a bump map for the texture as follows:



And then we will introduce the new specular map texture for specular light intensity such as follows:

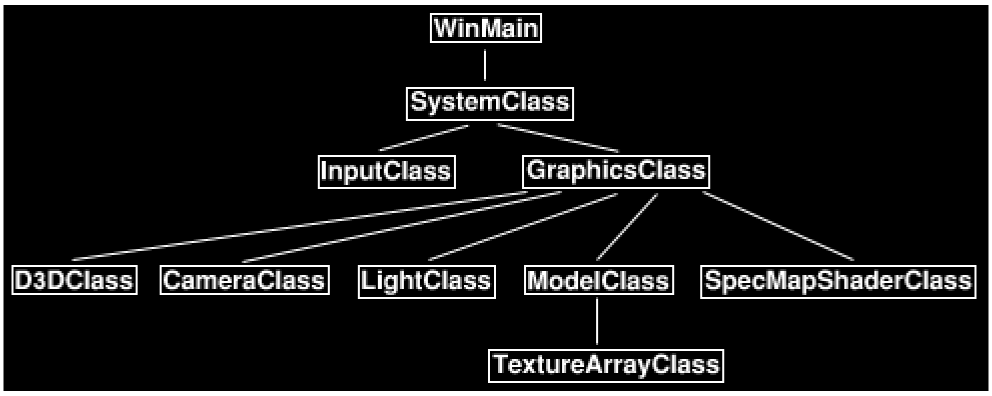


We use the grey scale in the specular map to determine the intensity of specular light at each pixel. Notice the image will allow each square on the base image to have its own specular intensity. And since the image is bump mapped it will highlight the bumps giving a very realistic surface such as the following:



Framework

The frame work has only changed slightly since the previous tutorial. It now has a SpecMapShaderClass instead of a BumpMapShaderClass.



We will start the tutorial by looking at the specular map HLSL shader code:

Specmap.vs

```
////////////////////////////////////////////////////////////////////////////////
// Filename: specmap.vs
////////////////////////////////////////////////////////////////////////////////


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

This shader uses specular lighting so we will need the camera position for calculating the view direction.

```
cbuffer CameraBuffer
{
	float3 cameraPosition;
};


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

The PixelInputType now has a viewDirection for specular lighting calculations.

```
struct PixelInputType
{
	float4 position : SV_POSITION;
	float2 tex : TEXCOORD0;
	float3 normal : NORMAL;
	float3 tangent : TANGENT;
	float3 binormal : BINORMAL;
	float3 viewDirection : TEXCOORD1;
};


////////////////////////////////////////////////////////////////////////////////
// Vertex Shader
////////////////////////////////////////////////////////////////////////////////
PixelInputType SpecMapVertexShader(VertexInputType input)
{
	PixelInputType output;
	float4 worldPosition;


	// 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);

    // 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);
```

For specular lighting calculations we will need the view direction calculated and passed into the pixel shader.

```
    // Calculate the position of the vertex in the world.
    worldPosition = mul(input.position, worldMatrix);

    // Determine the viewing direction based on the position of the camera and the position of the vertex in the world.
    output.viewDirection = cameraPosition.xyz - worldPosition.xyz;

    // Normalize the viewing direction vector.
    output.viewDirection = normalize(output.viewDirection);

    return output;
}
```

Specmap.ps

```
////////////////////////////////////////////////////////////////////////////////
// Filename: specmap.ps
////////////////////////////////////////////////////////////////////////////////


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

The specular map shader requires three textures in the texture array. The first texture is the color texture. The second texture is the normal map texture. And the third texture is the specular map texture.

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

This shader uses specular lighting so we will need the specular light color and power.

```
cbuffer LightBuffer
{
    float4 diffuseColor;
    float4 specularColor;
    float specularPower;
    float3 lightDirection;
};
```

```
//////////////
// TYPEDEFS //
//////////////
struct PixelInputType
{
	float4 position : SV_POSITION;
	float2 tex : TEXCOORD0;
	float3 normal : NORMAL;
	float3 tangent : TANGENT;
	float3 binormal : BINORMAL;
	float3 viewDirection : TEXCOORD1;
};


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

The first part of the shader is the regular bump map shader code.

```
	// 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 = input.normal + bumpMap.x * input.tangent + bumpMap.y * input.binormal;

	// 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;
```

If the light intensity is greater than zero we will do specular light calculations.

```
	if(lightIntensity > 0.0f)
	{
```

Here is where we sample the specular map for the intensity of specular light at this pixel.

```
		// Sample the pixel from the specular map texture.
		specularIntensity = shaderTextures[2].Sample(SampleType, input.tex);
```

In the reflection calculation we use the bump map normal instead of the regular input normal.

```
		// Calculate the reflection vector based on the light intensity, normal vector, and light direction.
		reflection = normalize(2 * lightIntensity * bumpNormal - lightDir);

		// Determine the amount of specular light based on the reflection vector, viewing direction, and specular
power.
		specular = pow(saturate(dot(reflection, input.viewDirection)), specularPower);
```

Now that we have the amount of specular light at this pixel we then multiply it by the specular intensity from the specular map to get a final value.

```
		// Use the specular map to determine the intensity of specular light at this pixel.
		specular = specular * specularIntensity;

		// Add the specular component last to the output color.
		color = saturate(color + specular);
	}

	return color;
}
```

Specmapshaderclass.h

The SpecMapShaderClass is just the BumpMapShaderClass modified slightly from the previous tutorial.

```
////////////////////////////////////////////////////////////////////////////////
// Filename: specmapshaderclass.h
////////////////////////////////////////////////////////////////////////////////
#ifndef _SPECMAPSHADERCLASS_H_
#define _SPECMAPSHADERCLASS_H_


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


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

We will need structures for the light buffer and camera buffer.

```
	struct LightBufferType
	{
		D3DXVECTOR4 diffuseColor;
		D3DXVECTOR4 specularColor;
		float specularPower;
		D3DXVECTOR3 lightDirection;
	};

	struct CameraBufferType
	{
		D3DXVECTOR3 cameraPosition;
		float padding;
	};

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

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

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

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

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

We have added the camera direction and light information buffers for the specular light calculations that will occur in the HLSL shader.

```
	ID3D11Buffer* m_lightBuffer;
	ID3D11Buffer* m_cameraBuffer;
};

#endif
```

Specmapshaderclass.cpp

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


SpecMapShaderClass::SpecMapShaderClass()
```

```
{
	m_vertexShader = 0;
	m_pixelShader = 0;
	m_layout = 0;
	m_matrixBuffer = 0;
	m_sampleState = 0;
```

Initialize the light and camera buffer to null in the class constructor.

```
	m_lightBuffer = 0;
	m_cameraBuffer = 0;
}


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


SpecMapShaderClass::~SpecMapShaderClass()
{
}


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

We load the new specmap.vs and specmap.ps HLSL shader files.

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

	return true;
}


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

	return;
}
```

The Render function now takes in the camera position, specular light color, and specular power for the specular lighting calculations that will be done by the shader.

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


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

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

	return true;
}


bool SpecMapShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR* 
psFilename)
{
	HRESULT result;
	ID3D10Blob* errorMessage;
	ID3D10Blob* vertexShaderBuffer;
	ID3D10Blob* pixelShaderBuffer;
	D3D11_INPUT_ELEMENT_DESC polygonLayout[5];
	unsigned int numElements;
	D3D11_BUFFER_DESC matrixBufferDesc;
	D3D11_SAMPLER_DESC samplerDesc;
	D3D11_BUFFER_DESC lightBufferDesc;
	D3D11_BUFFER_DESC cameraBufferDesc;


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

Here we load the specular map vertex shader.

```
	// Compile the vertex shader code.
	result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "SpecMapVertexShader", "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;
	}
```

Here we load the specular map pixel shader.

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

	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 and camera buffers are 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;
	}

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

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

	return true;
}


void SpecMapShaderClass::ShutdownShader()
{
```

The new light and camera buffers related to specular light are released here in the ShutdownShader function.

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

	// 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;
}


void SpecMapShaderClass::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;
}
```

SetShaderParameters now takes in the camera position, specular light color, and specular power.

```
bool SpecMapShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX 
worldMatrix,
                                                    D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, 
                                                    ID3D11ShaderResourceView** textureArray, 
D3DXVECTOR3 lightDirection,
                                                    D3DXVECTOR4 diffuseColor, D3DXVECTOR3 
cameraPosition, D3DXVECTOR4 specularColor,
                                                    float specularPower)
{
	HRESULT result;
	D3D11_MAPPED_SUBRESOURCE mappedResource;
	MatrixBufferType* dataPtr;
	unsigned int bufferNumber;
	LightBufferType* dataPtr2;
	CameraBufferType* dataPtr3;


	// 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 color, normal map, and spec map texture in the texture array are set inside the pixel shader here.

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

The light buffer is set here.

```
	// 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;
	dataPtr2->specularColor = specularColor;
	dataPtr2->specularPower = specularPower;

	// 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);
```

The camera buffer is set here.

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

	// Get a pointer to the data in the constant buffer.
	dataPtr3 = (CameraBufferType*)mappedResource.pData;

	// Copy the camera position into the constant buffer.
	dataPtr3->cameraPosition = cameraPosition;

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

	// Set the position of the camera constant buffer in the vertex shader as the second buffer.
	bufferNumber = 1;

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

	return true;
}


void SpecMapShaderClass::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 SpecMapShaderClass header is included here in the GraphicsClass header file.

```
#include "specmapshaderclass.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;
```

We create a new object for the SpecMapShaderClass.

```
	SpecMapShaderClass* m_SpecMapShader;
	LightClass* m_Light;
};

#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;
```

We initialize the SpecMapShaderClass object to null here in the class constructor.

```
	m_SpecMapShader = 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 model object loads the cube model, the stone02.dds color texture, the bump02.dds normal map, and the spec02.dds specular map. The three textures will be loaded into a three element texture array inside the ModelClass object.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), "../Engine/data/cube.txt", L"../Engine/data/stone02.dds",
					L"../Engine/data/bump02.dds", L"../Engine/data/spec02.dds");
	if(!result)
	{
		MessageBox(hwnd, L"Could not initialize the model object.", L"Error", MB_OK);
		return false;
	}
```

The new SpecMapShaderClass is created and initialized here.

```
	// Create the specular map shader object.
	m_SpecMapShader = new SpecMapShaderClass;
	if(!m_SpecMapShader)
	{
		return false;
	}

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

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

The specular components of the light are set for this tutorial.

```
	// Initialize the light object.
	m_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);
	m_Light->SetDirection(0.0f, 0.0f, 1.0f);
	m_Light->SetSpecularColor(1.0f, 1.0f, 1.0f, 1.0f);
	m_Light->SetSpecularPower(16.0f);

	return true;
}


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

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

```
	// Release the specular map shader object.
	if(m_SpecMapShader)
	{
		m_SpecMapShader->Shutdown();
		delete m_SpecMapShader;
		m_SpecMapShader = 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);

	// 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());
```

The specular map shader is used to render the cube model.

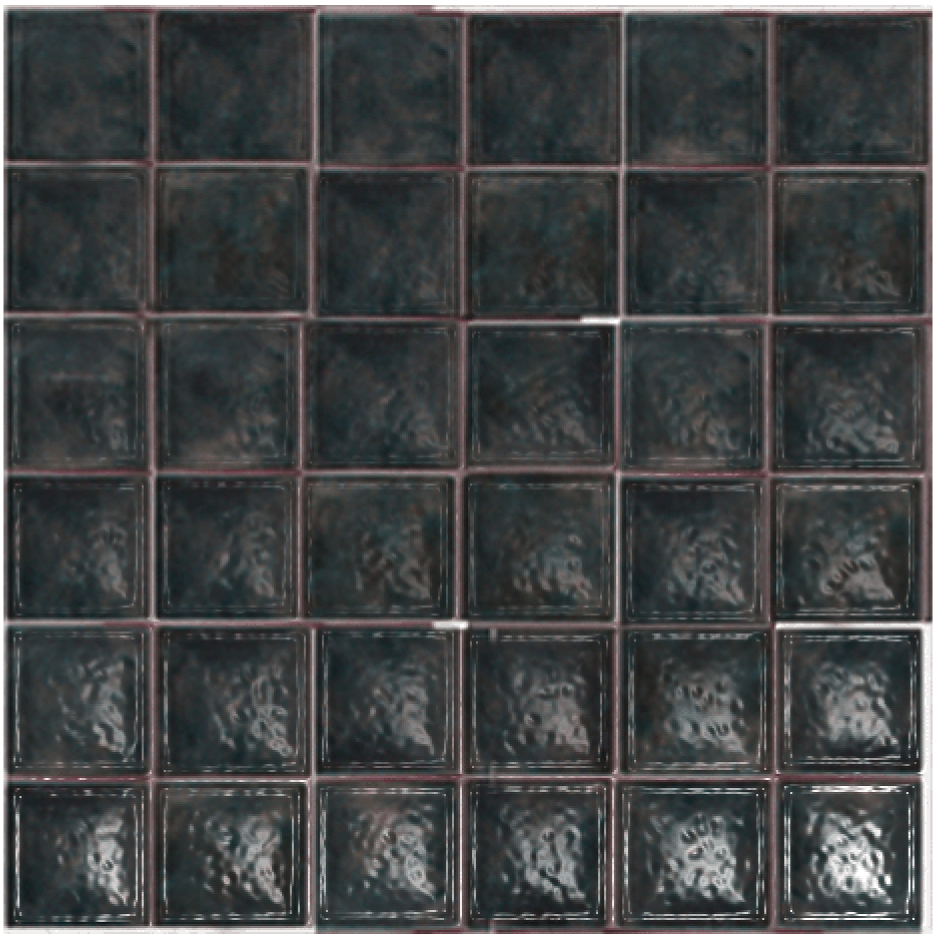
```
	// Render the model using the specular map shader.
	m_SpecMapShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
viewMatrix, projectionMatrix,
				    m_Model->GetTextureArray(), m_Light->GetDirection(), m_Light-
>GetDiffuseColor(),
				    m_Camera->GetPosition(), m_Light->GetSpecularColor(), m_Light-
>GetSpecularPower());

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

```
		return true;
}
```

Summary

With specular mapping we can control specular lighting on a per-pixel basis giving us the ability to create very unique highlighting effects.



To Do Exercises

1. Recompile the code and ensure you get a rotating cube with specular highlighting on the bump mapped surface. Press escape when done.

2. Create you own different specular maps to see how it changes the effect.

3. Change the pixel shader to return just the specular result.

4. Change the shininess of the light object in the GraphicsClass to see the change in effect.

[Original script: www.rastertek.com]