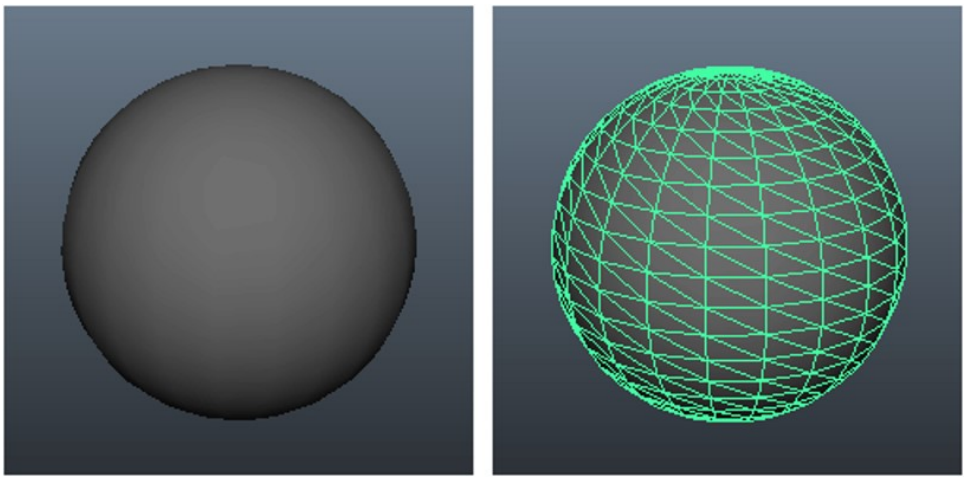
Tutorial: Buffers, Shaders, and HLSL

This tutorial will be the introduction to writing **vertex** and **index buffers** in DirectX 11. It will also be the introduction to using **vertex** and **pixel shaders** in DirectX 11. These are the most fundamental concepts that you need to understand and utilize to render 3D graphics.

Vertex Buffers

The first concept to understand is vertex buffers. To illustrate this concept let us take the example of a 3D model of a sphere, which is composed of hundreds of triangles (**polygons**):



Each of the triangles in the sphere model has three points to it, we call each point a vertex. So, for us to render the sphere model we need to put all the vertices that form the sphere into a special data array that we call a vertex buffer. Once all the points of the sphere model are in the vertex buffer, we can then send the vertex buffer to the GPU so that it can render the model.

Index Buffers

Index buffers are related to vertex buffers. Their purpose is to record the location of each vertex that is in the vertex buffer. The GPU then uses the index buffer to quickly find specific vertices in the vertex buffer. The concept of an index buffer is similar to the concept using an index in a book, it helps find the topic you are looking for at a much higher speed. The DirectX SDK documentation says that using index buffers can also increase the possibility of caching the vertex data in faster locations in video memory. So, it is highly advised to use these for performance reasons as well.

Vertex Shaders

Vertex shaders are small programs that are written mainly for transforming the vertices from the vertex buffer into 3D space. There are other calculations that can be done such as calculating **normals** for each vertex. The vertex shader program will be called by the GPU for each vertex it needs to process. For example, a 5,000 polygon model will run your vertex shader program 15,000 times each frame just to draw that single model. So, if you lock your graphics program to 60 fps it will call your vertex shader 900,000 times a second to draw just 5,000 triangles. As you can tell writing efficient vertex shaders is important.

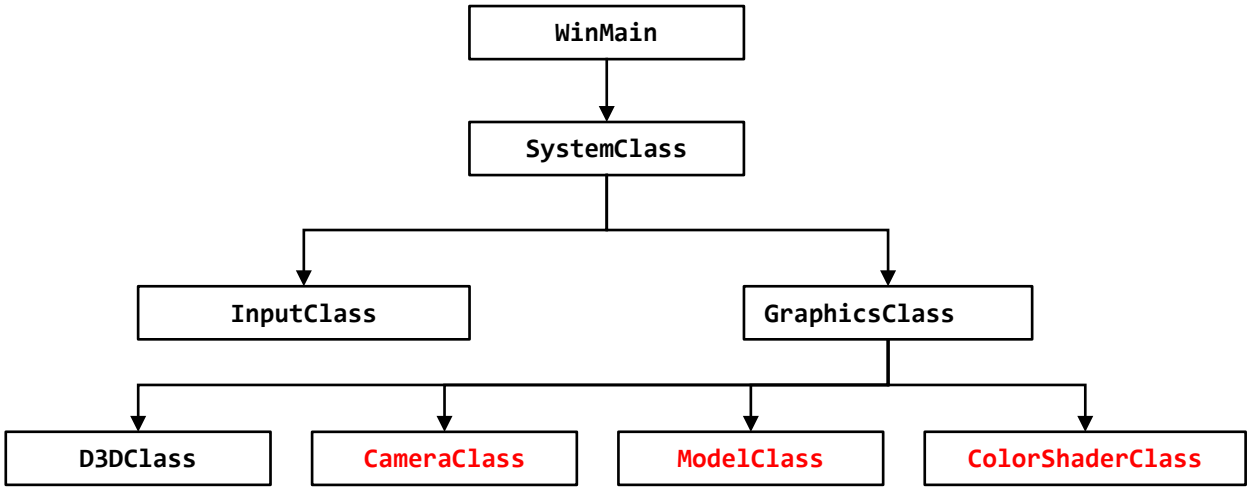
Pixel Shaders

Pixel shaders are small programs that are written for doing the coloring of the polygons that we draw. They are run by the GPU for every visible pixel that will be drawn to the screen. Coloring, texturing, lighting, and most other affects you plan to do to your polygon faces are handled by the pixel shader program. Pixel shaders must be efficiently written due to the number of times they will be called by the GPU.

HLSL

High-Level Shader Language (HLSL) is the language we use in DirectX 11 to code these small vertex and pixel shader programs. The syntax is pretty much identical to the C language with some pre-defined types. HLSL program files are composed of global variables, type defines, vertex shaders, pixel shaders, and geometry shaders. As this is the first HLSL tutorial we will do a very simple HLSL program using DirectX 11 to get started.

Framework



- CameraClass takes care of our view matrix, which handle the location of the camera in the world and pass it to shaders when they need to draw and figure out where we are looking at the scene from.
- ModelClass handles the geometry of our 3D models.
- ColorShaderClass will be responsible for rendering the model to the screen invoking our HLSL shader.

We will begin the tutorial code by looking at the HLSL shader programs first.

color.vs

These will be our first shader programs. Shaders are small programs that do the actual rendering of models. These shaders are written in HLSL and stored in source files called color.vs and color.ps. We placed the files with the .cpp and .h files in the engine for now. The purpose of this shader is just to draw colored triangles as we're keeping things simple as possible in this first HLSL tutorial. Here is the code for the vertex shader first:

In shader programs you begin with the global variables. These globals can be modified externally from your C++ code. You can use many types of variables such as int or float and then set them externally for the shader program to use. Generally, you will put most globals in buffer object types called "cbuffer" even if it is just a single global variable. Logically organizing these buffers is important for efficient execution of shaders as well as how the graphics card will store the buffers. In this example, we've put three matrices in the same buffer since we will update them each frame at the same time.

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

Similar to C we can create our own type definitions. We will use different types such as float4 that are available to HLSL which make programming shaders easier and readable. In this example, we are creating types that have x, y, z, w position vectors and red, green, blue, alpha colors. The POSITION, COLOR, and SV_POSITION are **semantics** that convey to the GPU the use of the variable. We have to create two different structures here since the semantics are different for vertex and pixel shaders even though the structures are the same otherwise. POSITION works for vertex shaders and SV_POSITION works for pixel shaders while COLOR works for both. If you want more than one of the same types, then you have to add a number to the end such as COLOR0, COLOR1, and so forth.

```
// Type definitions
struct VertexInputType
{
    float4 position : POSITION;
    float4 color : COLOR;
};

struct PixelInputType
{
```

```
    float4 position : SV_POSITION;
    float4 color : COLOR;
};
```

The vertex shader is called by the GPU when it is processing data from the vertex buffers that have been sent to it. This vertex shader which We named ColorVertexShader will be called for every single vertex in the vertex buffer. The input to the vertex shader must match the data format in the vertex buffer as well as the type definition in the shader source file which in this case is VertexInputType. The output of the vertex shader will be sent to the pixel shader. In this case the output type is called PixelInputType which is defined above as well.

With that in mind you see that the vertex shader creates an output variable that is of the PixelInputType type. It then takes the position of the input vertex and multiplies it by the world, view, and then projection matrices. This will place the vertex in the correct location for rendering in 3D space according to our view and then onto the 2D screen. After that the output variable takes a copy of the input color and then returns the output which will be used as input to the pixel shader. Also note that we do set the W value of the input position to 1.0; otherwise, it is undefined since we only read in a XYZ vector for position.

```
// Vertex Shader
PixelInputType ColorVertexShader(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 input color for the pixel shader to use.
    output.color = input.color;

    return output;
}
```

color.ps

The pixel shader draws each pixel on the polygons that will be rendered to the screen. In this pixel shader, it uses PixelInputType as input and returns a float4 as output which represents the final pixel color. This pixel shader program is very simple as we just tell it to color the pixel the same as the input value of the color. Note that the pixel shader gets its input from the vertex shader output.

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


// Pixel Shader
float4 ColorPixelShader(PixelInputType input) : SV_TARGET
{
    return input.color;
}
```

modelclass.h

As stated previously the ModelClass is responsible for encapsulating the **geometry** for 3D models. In this tutorial, we will manually setup the data for a single green triangle. We will also create a vertex and index buffer for the triangle so that it can be rendered.

```
#ifndef _MODELCLASS_H_
#define _MODELCLASS_H_

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

class ModelClass
{
```

Here is the definition of our vertex type that will be used with the vertex buffer in this ModelClass. Also take note that this typedef must match the layout in the ColorShaderClass that will be looked at later in the tutorial.

```
private:
	struct VertexType
	{
		D3DXVECTOR3 position;
		D3DXVECTOR4 color;
	};

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

The functions here handle initializing and shutdown of the model's vertex and index buffers. The Render function puts the model geometry on the video card to prepare it for drawing by the color shader.

```
	bool Initialize(ID3D11Device*);
	void Shutdown();
	void Render(ID3D11DeviceContext*);

	int GetIndexCount();

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

The private variables in the ModelClass are the vertex and index buffer as well as two integers to keep track of the size of each buffer. Note that all DirectX 11 buffers generally use the generic ID3D11Buffer type and are more clearly identified by a buffer description when they are first created.

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

#endif
```

modelclass.cpp

```
#include "modelclass.h"
```

The class constructor initializes the vertex and index buffer pointers to null.

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

ModelClass::ModelClass(const ModelClass& other)
{
}
```

```
ModelClass::~ModelClass()
{
}
```

The Initialize function will call the initialization functions for the vertex and index buffers.

```
bool ModelClass::Initialize(ID3D11Device* device)
{
	bool result;

	// Initialize the vertex and index buffer that hold the geometry for the triangle.
	result = InitializeBuffers(device);
	if(!result)
	{
		return false;
	}

	return true;
}
```

The Shutdown function will call the shutdown functions for the vertex and index buffers.

```
void ModelClass::Shutdown()
{
	// Release the vertex and index buffers.
	ShutdownBuffers();

	return;
}
```

Render is called from the GraphicsClass::Render function. This function calls RenderBuffers to put the vertex and index buffers on the graphics pipeline so the color shader will be able to render them.

```
void ModelClass::Render(ID3D11DeviceContext* deviceContext)
{
	// Put the vertex and index buffers on the graphics pipeline to prepare them for drawing.
	RenderBuffers(deviceContext);

	return;
}
```

GetIndexCount returns the number of indexes in the model. The color shader will need this information to draw this model.

```
int ModelClass::GetIndexCount()
{
	return m_indexCount;
}
```

The InitializeBuffers function is where we handle creating the vertex and index buffers. Usually you would read in a model and create the buffers from that data file. For this tutorial, we will just set the points in the vertex and index buffer manually since it is only a single triangle.

```
bool ModelClass::InitializeBuffers(ID3D11Device* device)
{
	VertexType* vertices;
	unsigned long* indices;
	D3D11_BUFFER_DESC vertexBufferDesc, indexBufferDesc;
	D3D11_SUBRESOURCE_DATA vertexData, indexData;
	HRESULT result;
```

First create two temporary arrays to hold the vertex and index data that we will use later to populate the final buffers with.

```
// Set the number of vertices in the vertex array.
m_vertexCount = 3;

// Set the number of indices in the index array.
m_indexCount = 3;

// 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;
}
```

Now fill both the vertex and index array with the three points of the triangle as well as the index to each of the points. Please note that we create the points in the clockwise order of drawing them. If you do this counterclockwise it will think the triangle is facing the opposite direction and not draw it due to **back face culling**. Always remember that the order in which you send your vertices to the GPU is very important. The color is set here as well since it is part of the vertex description. We set the color to green.

```
// Load the vertex array with data.
vertices[0].position = D3DXVECTOR3(-1.0f, -1.0f, 0.0f);  // Bottom left.
vertices[0].color = D3DXVECTOR4(0.0f, 1.0f, 0.0f, 1.0f);

vertices[1].position = D3DXVECTOR3(0.0f, 1.0f, 0.0f);  // Top middle.
vertices[1].color = D3DXVECTOR4(0.0f, 1.0f, 0.0f, 1.0f);

vertices[2].position = D3DXVECTOR3(1.0f, -1.0f, 0.0f);  // Bottom right.
vertices[2].color = D3DXVECTOR4(0.0f, 1.0f, 0.0f, 1.0f);

// Load the index array with data.
indices[0] = 0;  // Bottom left.
indices[1] = 1;  // Top middle.
indices[2] = 2;  // Bottom right.
```

With the vertex array and index array filled out, we can now use those to create the vertex buffer and index buffer. Creating both buffers is done in the same fashion. First, fill out a description of the buffer. In the description the ByteWidth (size of the buffer) and the BindFlags (type of buffer) are what you need to ensure are filled out correctly. After the description is filled out you need to also fill out a subresource pointer which will point to either your vertex or index array you previously created. With the description and subresource pointer you can call CreateBuffer using the D3D device and it will return a pointer to your new buffer.

```
// 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;
	}
```

After the vertex buffer and index buffer have been created you can delete the vertex and index arrays as they are no longer needed since the data was copied into the buffers.

```
	// 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;
}
```

The ShutdownBuffers function just releases the vertex buffer and index buffer that were created in the InitializeBuffers function.

```
void ModelClass::ShutdownBuffers()
{
	// Release the index buffer.
	if(m_indexBuffer)
	{
		m_indexBuffer->Release();
		m_indexBuffer = 0;
	}

	// Release the vertex buffer.
	if(m_vertexBuffer)
	{
		m_vertexBuffer->Release();
		m_vertexBuffer = 0;
	}

	return;
}
```

RenderBuffers is called from the Render function. The purpose of this function is to set the vertex buffer and index buffer as active on the **input assembler** in the GPU. Once the GPU has an active vertex buffer, it can then use the shader to render that buffer. This function also defines how those buffers should be drawn such as triangles, lines, fans, and so forth. In this tutorial, we set the vertex buffer and index buffer as active on the input assembler and tell the GPU that the buffers should be drawn as triangles using the IASetPrimitiveTopology DirectX function.

```
void ModelClass::RenderBuffers(ID3D11DeviceContext* deviceContext)
{
```

```
	unsigned int stride;
	unsigned int offset;

	// Set vertex buffer stride and offset.
	stride = sizeof(VertexType);
	offset = 0;

	// Set the vertex buffer to active in the input assembler so it can be rendered.
	deviceContext->IASetVertexBuffers(0, 1, &m_vertexBuffer, &stride, &offset);

	// Set the index buffer to active in the input assembler so it can be rendered.
	deviceContext->IASetIndexBuffer(m_indexBuffer, DXGI_FORMAT_R32_UINT, 0);

	// Set the type of primitive that should be rendered from this vertex buffer, in this case triangles.
	deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);

	return;
}
```

colorshaderclass.h

The ColorShaderClass is what we will use to invoke our HLSL shaders for drawing the 3D models that are on the GPU.

```
#ifndef _COLORSHADERCLASS_H_
#define _COLORSHADERCLASS_H_

#include <d3d11.h>
#include <d3dx10math.h>
#include <d3dx11async.h>
#include <fstream>

using namespace std;

class ColorShaderClass
{
private:
```

Here is the definition of the cBuffer type that will be used with the vertex shader. This typedef must be exactly the same as the one in the vertex shader as the model data needs to match the typedefs in the shader for proper rendering.

```
	struct MatrixBufferType
	{
		D3DXMATRIX world;
		D3DXMATRIX view;
		D3DXMATRIX projection;
	};

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

The functions here handle initializing and shutdown of the shader. The render function sets the shader parameters and then draws the prepared model vertices using the shader.

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

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

	bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX);
```

```
	void RenderShader(ID3D11DeviceContext*, int);

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

#endif
```

colorshaderclass.cpp

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

As usual the class constructor initializes all the private pointers in the class to null.

```
ColorShaderClass::ColorShaderClass()
{
	m_vertexShader = 0;
	m_pixelShader = 0;
	m_layout = 0;
	m_matrixBuffer = 0;
}

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

ColorShaderClass::~ColorShaderClass()
{
}
```

The Initialize function will call the initialization function for the shaders. We pass in the name of the HLSL shader files, in this tutorial they are named color.vs and color.ps.

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

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

	return true;
}
```

The Shutdown function will call the shutdown of the shader.

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

	return;
}
```

Render will first set the parameters inside the shader using the SetShaderParameters function. Once the parameters are set it then calls RenderShader to draw the green triangle using the HLSL shader.

```
bool ColorShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
```

```
        D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix)
{
        bool result;

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

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

        return true;
}
```

Now, we will start with one of the more important functions to this tutorial which is called InitializeShader. This function is what actually loads the shader files and makes it usable to DirectX and the GPU. You will also see the setup of the layout and how the vertex buffer data is going to look on the graphics pipeline in the GPU. The layout will need the match the VertexType in the modelclass.h file as well as the one defined in the color.vs file.

```
bool ColorShaderClass::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;

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

Here is where we compile the shader programs into buffers. We give it the name of the shader file, the name of the shader, the shader version (5.0 in DirectX 11), and the buffer to compile the shader into. If it fails compiling the shader, it will put an error message inside the errorMessage string which we send to another function to write out the error. If it still fails and there is no errorMessage string, then it means it could not find the shader file in which case we pop up a dialog box saying so.

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

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

Once the vertex shader and pixel shader code has successfully compiled into buffers, we then use those buffers to create the shader objects themselves. We will use these pointers to interface with the vertex and pixel shader from this point forward.

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

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

The next step is to create the layout of the vertex data that will be processed by the shader. As this shader uses a position and color vector, we need to create both in the layout specifying the size of both. The semantic name is the first thing to fill out in the layout, this allows the shader to determine the usage of this element of the layout. As we have two different elements, we use POSITION for the first one and COLOR for the second. The next important part of the layout is the Format. For the position vector, we use DXGI_FORMAT_R32G32B32_FLOAT and for the color we use DXGI_FORMAT_R32G32B32A32_FLOAT. The final thing you need to pay attention to is the AlignedByteOffset which indicates how the data is spaced in the buffer. For this layout, we are telling it the first 12 bytes are position and the next 16 bytes will be color, AlignedByteOffset shows where each element begins. You can use D3D11_APPEND_ALIGNED_ELEMENT instead of placing your own values in AlignedByteOffset and it will figure out the spacing for you. The other settings we've made default for now as they are not needed in this tutorial.

```
// Now setup the layout of the data that goes into the shader.
// 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 = "COLOR";
polygonLayout[1].SemanticIndex = 0;
polygonLayout[1].Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
polygonLayout[1].InputSlot = 0;
polygonLayout[1].AlignedByteOffset = D3D11_APPEND_ALIGNED_ELEMENT;
polygonLayout[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
polygonLayout[1].InstanceDataStepRate = 0;
```

Once the layout description has been setup, we can get the size of it and then create the input layout using the D3D device. Also release the vertex and pixel shader buffers since they are no longer needed once the layout has been created.

```
	// 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;
```

The final thing that needs to be setup to utilize the shader is the constant buffer. As you saw in the vertex shader, we currently have just one constant buffer so we only need to setup one here so we can interface with the shader. The buffer usage needs to be set to dynamic since we will be updating it each frame. The bind flags indicate that this buffer will be a constant buffer. The cpu access flags need to match up with the usage so it is set to D3D11_CPU_ACCESS_WRITE. Once we fill out the description, we can then create the constant buffer interface and then use that to access the internal variables in the shader using the function SetShaderParameters.

```
	// Setup the description of the dynamic matrix 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 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;
	}

	return true;
}
```

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

```
void ColorShaderClass::ShutdownShader()
{
	// 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;
}
```

The OutputShaderErrorMessage writes out error messages that are generating when compiling either vertex shaders or pixel shaders.

```
void ColorShaderClass::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 SetShaderVariables function exists to make setting the global variables in the shader easier. The matrices used in this function are created inside the GraphicsClass, after which this function is called to send them from there into the vertex shader during the Render function call.

```
bool ColorShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix)
{
	HRESULT result;
	D3D11_MAPPED_SUBRESOURCE mappedResource;
	MatrixBufferType* dataPtr;
```

```
	unsigned int bufferNumber;
```

Make sure to transpose matrices before sending them into the shader, this is a requirement for DirectX 11.

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

Lock the m_matrixBuffer, set the new matrices inside it, and then unlock it.

```
	// Lock the 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 constant buffer.
	deviceContext->Unmap(m_matrixBuffer, 0);
```

Now set the updated matrix buffer in the HLSL vertex shader.

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

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

	return true;
}
```

RenderShader is the second function called in the Render function. SetShaderParameters is called before this to ensure the shader parameters are setup correctly.

The first step in this function is to set our input layout to active in the input assembler. This lets the GPU know the format of the data in the vertex buffer. The second step is to set the vertex shader and pixel shader we will be using to render this vertex buffer. Once the shaders are set, we render the triangle by calling the DrawIndexed DirectX 11 function using the D3D device context. Once this function is called it will render the green triangle.

```
void ColorShaderClass::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);

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

	return;
}
```

cameraclass.h

We have examined how to code HLSL shaders, how to setup vertex and index buffers, and how to invoke the HLSL shaders to draw those buffers using the ColorShaderClass. The one thing we are missing however is the view point to draw them from. For this, we will require a camera class to let DirectX 11 know from where and also how we are viewing the scene. The camera class will keep track of where the camera is and its current rotation. It will use the position and rotation information to generate a view matrix which will be passed into the HLSL shader for rendering.

```
#ifndef _CAMERACLASS_H_
#define _CAMERACLASS_H_

#include <d3dx10math.h>

class CameraClass
{
public:
	CameraClass();
	CameraClass(const CameraClass&);
	~CameraClass();

	void SetPosition(float, float, float);
	void SetRotation(float, float, float);

	D3DXVECTOR3 GetPosition();
	D3DXVECTOR3 GetRotation();

	void Render();
	void GetViewMatrix(D3DXMATRIX&);

private:
	float m_positionX, m_positionY, m_positionZ;
	float m_rotationX, m_rotationY, m_rotationZ;
	D3DXMATRIX m_viewMatrix;
};

#endif
```

The CameraClass header is quite simple with just four functions that will be used. The SetPosition and SetRotation functions will be used to set the position and rotation of the camera object. Render will be used to create the view matrix based on the position and rotation of the camera. And finally, GetViewMatrix will be used to retrieve the view matrix from the camera object so that the shaders can use it for rendering.

cameraclass.cpp

```
#include "cameraclass.h"
```

The class constructor will initialize the position and rotation of the camera to be at the origin of the scene.

```
CameraClass::CameraClass()
{
	m_positionX = 0.0f;
	m_positionY = 0.0f;
	m_positionZ = 0.0f;

	m_rotationX = 0.0f;
	m_rotationY = 0.0f;
	m_rotationZ = 0.0f;
}

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

CameraClass::~CameraClass()
{
}
```

The SetPosition and SetRotation functions are used for setting up the position and rotation of the camera.

```
void CameraClass::SetPosition(float x, float y, float z)
{
	m_positionX = x;
	m_positionY = y;
	m_positionZ = z;
	return;
}

void CameraClass::SetRotation(float x, float y, float z)
{
	m_rotationX = x;
	m_rotationY = y;
	m_rotationZ = z;
	return;
}
```

The GetPosition and GetRotation functions return the location and rotation of the camera to calling functions.

```
D3DXVECTOR3 CameraClass::GetPosition()
{
	return D3DXVECTOR3(m_positionX, m_positionY, m_positionZ);
}

D3DXVECTOR3 CameraClass::GetRotation()
{
	return D3DXVECTOR3(m_rotationX, m_rotationY, m_rotationZ);
}
```

The Render function uses the position and rotation of the camera to build and update the view matrix. We first setup our variables for up, position, rotation, and so forth. Then at the origin of the scene we first rotate the camera based on the x, y, and z rotation of the camera. Once it is properly rotated when then translate the camera to the position in 3D space. With the correct values in the position, lookAt, and up we can then use the D3DXMatrixLookAtLH function to create the view matrix to represent the current camera rotation and translation.

```
void CameraClass::Render()
{
	D3DXVECTOR3 up, position, lookAt;
	float yaw, pitch, roll;
	D3DXMATRIX rotationMatrix;

	// Setup the vector that points upwards.
	up.x = 0.0f;
	up.y = 1.0f;
	up.z = 0.0f;

	// Setup the position of the camera in the world.
	position.x = m_positionX;
	position.y = m_positionY;
	position.z = m_positionZ;

	// Setup where the camera is looking by default.
	lookAt.x = 0.0f;
	lookAt.y = 0.0f;
	lookAt.z = 1.0f;

	// Set the yaw (Y axis), pitch (X axis), and roll (Z axis) rotations in radians.
	pitch = m_rotationX * 0.0174532925f;
	yaw   = m_rotationY * 0.0174532925f;
	roll  = m_rotationZ * 0.0174532925f;

	// Create the rotation matrix from the yaw, pitch, and roll values.
	D3DXMatrixRotationYawPitchRoll(&rotationMatrix, yaw, pitch, roll);
```

```
	// Transform the lookAt and up vector by the rotation matrix so the view is correctly rotated at the origin.
	D3DXVec3TransformCoord(&lookAt, &lookAt, &rotationMatrix);
	D3DXVec3TransformCoord(&up, &up, &rotationMatrix);

	// Translate the rotated camera position to the location of the viewer.
	lookAt = position + lookAt;

	// Finally create the view matrix from the three updated vectors.
	D3DXMatrixLookAtLH(&m_viewMatrix, &position, &lookAt, &up);

	return;
}
```

After the Render function has been called to create the view matrix, we can provide the updated view matrix to calling functions using this GetViewMatrix function. The view matrix will be one of the three main matrices used in the HLSL vertex shader.

```
void CameraClass::GetViewMatrix(D3DXMATRIX& viewMatrix)
{
	viewMatrix = m_viewMatrix;
	return;
}
```

graphicsclass.h

GraphicsClass now has the three new classes added to it. CameraClass, ModelClass, and ColorShaderClass have headers added here as well as private member variables. Remember that GraphicsClass is the main class that is used to render the scene by invoking all the needed class objects for the project.

```
#include "cameraclass.h"
#include "modelclass.h"
#include "colorshaderclass.h"

class GraphicsClass
{
private:
	CameraClass* m_Camera;
	ModelClass* m_Model;
	ColorShaderClass* m_ColorShader;
};
```

graphicsclass.cpp

The first change to GraphicsClass is initializing the camera, model, and color shader objects in the class constructor to null.

```
GraphicsClass::GraphicsClass()
{
	m_Camera = 0;
	m_Model = 0;
	m_ColorShader = 0;
}
```

The Initialize function has also been updated to create and initialize the three new objects.

```
bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
	// Create the camera object.
	m_Camera = new CameraClass;
	if(!m_Camera)
	{
		return false;
	}
```

```
	// Set the initial position of the camera.
	m_Camera->SetPosition(0.0f, 0.0f, -10.0f);

	// Create the model object.
	m_Model = new ModelClass;
	if(!m_Model)
	{
		return false;
	}

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

	// Create the color shader object.
	m_ColorShader = new ColorShaderClass;
	if(!m_ColorShader)
	{
		return false;
	}

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

	return true;
}
```

Shutdown is also updated to shutdown and release the three new objects.

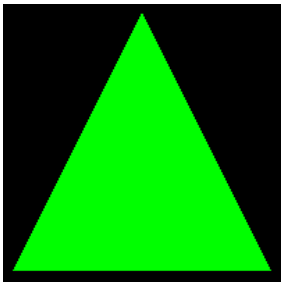
```
void GraphicsClass::Shutdown()
{
	// Release the color shader object.
	if(m_ColorShader)
	{
		m_ColorShader->Shutdown();
		delete m_ColorShader;
		m_ColorShader = 0;
	}

	// 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;
	}

	return;
}
```

The Frame function has remained the same as the previous tutorial.

As you would expect the Render function had the most changes to it. It still begins with clearing the scene except that it is cleared to black. After that it calls the Render function for the camera object to create a view matrix based on the camera's location that was set in the Initialize function. Once the view matrix is created, we get a copy of it from the camera class. We also get copies of the world and projection matrix from the D3DClass object. We then call the ModelClass::Render function to put the green triangle model geometry on the graphics pipeline. With the vertices now prepared we call the color shader to draw the vertices using the model information and the three matrices for positioning each vertex. The green triangle is now drawn to the back buffer. With that the scene is complete and we call EndScene to display it to the screen.

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

	// Clear the buffers to begin the scene.
	m_D3D->BeginScene(0.5f, 0.5f, 0.5f, 1.0f);
	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, and projection matrices from the camera and d3d objects.
	m_Camera->GetViewMatrix(viewMatrix);
	m_D3D->GetWorldMatrix(worldMatrix);
	m_D3D->GetProjectionMatrix(projectionMatrix);

	// 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 color shader.
	result = m_ColorShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
		viewMatrix, projectionMatrix);
	if(!result)
	{
		return false;
	}

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

	return true;
}
```

Summary



In summary, you should have learned the basics about how vertex and index buffers work. You should have also learned the basics of vertex and pixel shaders and how to write them using HLSL. And finally, you should understand how we've incorporated these new concepts into our framework to produce a green triangle that renders to the screen. We also want to mention that we realize the code is fairly long just to draw a single triangle and it could have all been stuck inside a single main() function. However, we did it this way with a proper framework so that the coming tutorials require very few changes in the code to do far more complex graphics.

Exercise

1. Add two more polygons with different shapes and colors around the triangle.
2. Move the camera so all polygons can be seen at the same time.
3. Change the pixel shader to output the color half as bright. (Hint: multiply something in ColorPixelShader)

[Original script: www.rastertek.com]