# Direct3D 11 with Windows 8 Metro Applications

Frank Luna
May 11, 2012
www.d3dcoder.net

Direct3D 11 is the official 3D rendering API for Windows 8 Metro styled applications. The API is the same for both desktop and Metro styled applications. Therefore, while the book *Introduction to 3D Game Programming with DirectX 11* is not Metro specific, all the core Direct3D concepts apply. However, the book's sample projects will not compile as *Metro* applications for a couple of reasons:

- The samples do not use the Metro API for creating windows (i.e., the sample applications were not written as Metro styled applications).

- The samples use the Effects framework and the D3DX library for texture loading, which are not supported by Metro styled applications.

The samples should compile and run just fine as desktop applications on Windows 8 because Windows 8 maintains compatibility with non-Metro styled applications. (Instructions for getting the book demos working on Win8 and Visual Studio 2012 (VS12) are available at http://d3dcoder.net/WordPress/?p=17 and also the forums http://www.d3dcoder.net/phpBB/viewforum.php?f=4).

This tutorial shows how to port the "Textured Columns" exercise from Chapter 8 to a Metro styled application. With this tutorial, you should feel comfortable applying the concepts in *Introduction to 3D Game Programming with DirectX 11* to Metro styled applications. We have three main topics to tackle:

- Managing constant buffers. In the book, we mostly let the Effects framework handle this for us. Because we cannot use the Effects framework with Metro styled applications, we must update constant buffers ourselves and bind them to the pipeline ourselves. As an added benefit, by managing constant buffers ourselves, we can potentially improve efficiency over the Effects framework.

- Create shader permutations. In the book, we used the Effects framework as a "shader generator" to generate different techniques based on compile time constants. This let us create specialized shaders easily. Without the Effects framework, we must either implement a "shader generator" ourselves, or we can use conditional compilation to achieve the same result (i.e., use macros to add/remove code).

- Load textures from file. In the book, we used D3DX to load textures. The D3DX library is not supported in Metro styled applications. Fortunately, the DirectX

team has provided texture loading libraries that work with Metro styled applications.

We are using the Windows 8 Pro, along with Visual Studio 2012 Express for Windows 8. Windows 8 introduces a new API called WinRT (Windows Run-Time) for writing Windows 8 Metro applications. This API replaces the Win32 API. This tutorial is not a WinRT tutorial, nor a tutorial on the WinRT C++ language (C++ extended to support WinRT concepts like properties, delegates, managed references, etc.). The reader should consult MSDN for documentation on these concepts. The new WinRT C++ language is very similar to the C++/CLI language, which was used to write managed .NET applications in a C++ like language. It is assumed the reader has a basic familiarity with Direct3D 11; for example, the reader has worked through at least Part II of the book *Introduction to 3D Game Programming with DirectX 11*.

## §1 Direct3D 11 Metro Project Template

This section gives a brief overview of creating a Direct3D 11 Metro styled project. Launch Visual Studio 2012 and choose **Direct3D Application** (see Figure 1).

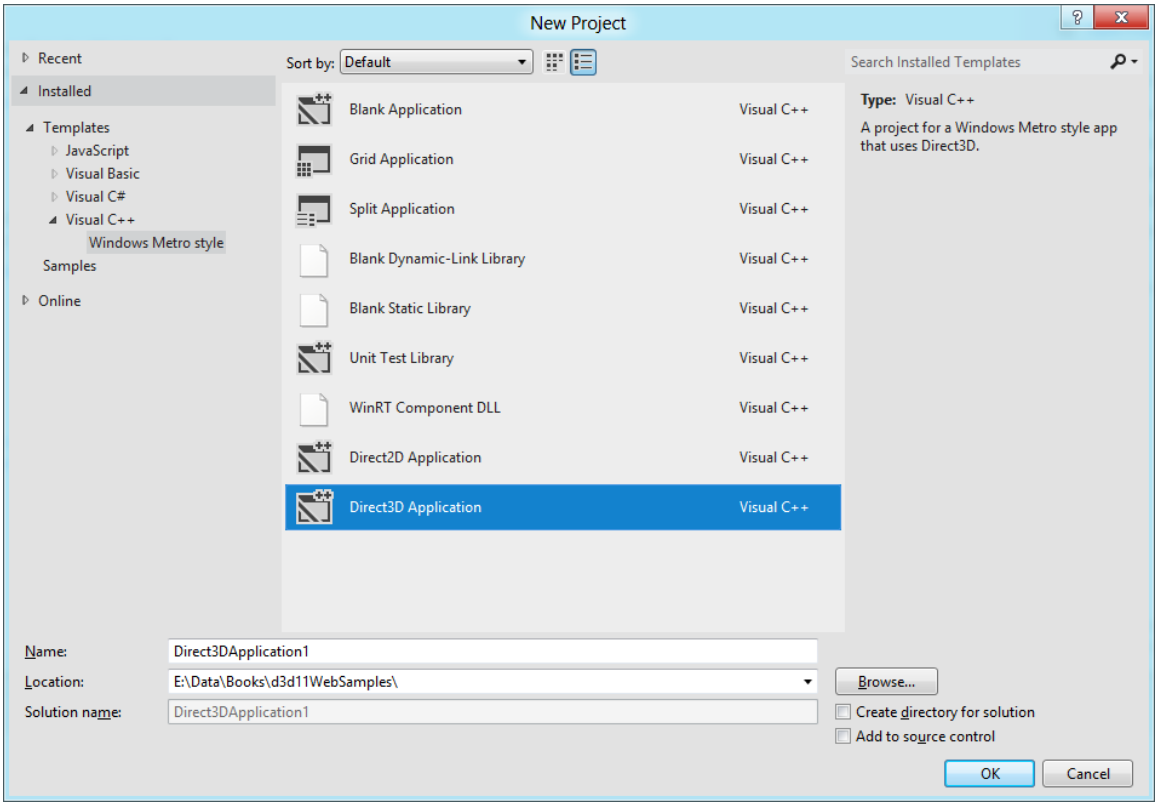


Figure 1

The Direct3D 11 project template creates a simple Metro app that renders a spinning colored cube. You can modify the code to build your real app. Before modifying the

code, however, you might want to build and run the program just to make sure everything is working. Below is a summary of the files the project creates:

| Files | Description |
|---|---|
| <d3d11Metro.h/.cpp (name varies—based on project name) | Implements the `IFrameworkView` interface. This interface contains several methods that are called during an app's creation process, which allows us to configure the app, allocate resource, and respond to app events such as when the app is suspended or the window is resized. Additionally, this interface contains a `Run` method, where we implement the message loop of our app. |
| Direct3DBase.h/.cpp | An abstract base class that handles initializing the Direct3D device and context, back buffer and depth buffer. This class is very similar to the `D3DApp` class that the book uses as a base class for all of the samples. |
| CubeRenderer.h/.cpp | Derives from `Direct3DBase` and implements application specific code for drawing a colored cube. This is analogous to how the book's samples derive from `D3DApp` and implement the specific sample code in the derived class. You probably want to rename this class based on your project. |
| BasicTimer.h/.cpp | Similar to the book's `GameTimer` class. |
| DirectXHelper.h/.cpp | Contains utility code such as throwing an exception for a failed HRESULT, and asynchronous data loading code. |
| SimpleVertexShader.hlsl SimplePixelShader.hlsl | Vertex and pixel shader the default sample uses for the colored cube. We can modify these or add new shaders. |
| Package.appxmanifest | Contains properties of your application you need to set when submitting application to market place, such as its description, logos, orientation support, and the device capabilities the application uses (e.g., access to microphone or GPS location). |

We will not go into the details of implementing the `IFrameworkView` class interface. The code is mostly self-explanatory and not Direct3D related. The reader may find the following links helpful for more information on the Metro app framework:

1. *Setting up the Game Project*
http://msdn.microsoft.com/en-us/library/windows/apps/hh780569.aspx

2. *Defining the Game's Metro Style App Framework*
http://msdn.microsoft.com/en-us/library/windows/apps/hh780566.aspx

3. *Creating a Great Metro Style Game for Windows 8* (GDC 2012 Presentation)
http://www.microsoft.com/en-us/download/details.aspx?id=29206

The `Direct3DBase` class is defined as follows:

```
#include "DirectXHelper.h"

// Helper class that initializes DirectX APIs for 3D rendering.
ref class Direct3DBase abstract
{
internal:
	Direct3DBase();

public:
	virtual void Initialize(Windows::UI::Core::CoreWindow^ window);
	virtual void HandleDeviceLost();
	virtual void CreateDeviceResources();
	virtual void CreateWindowSizeDependentResources();
	virtual void UpdateForWindowSizeChange();
	virtual void Render() = 0;
	virtual void Present();
	virtual float ConvertDipsToPixels(float dips);

protected private:
	// Direct3D Objects.
	Microsoft::WRL::ComPtr<ID3D11Device1> m_d3dDevice;
	Microsoft::WRL::ComPtr<ID3D11DeviceContext1> m_d3dContext;
	Microsoft::WRL::ComPtr<IDXGISwapChain1> m_swapChain;
	Microsoft::WRL::ComPtr<ID3D11RenderTargetView> m_renderTargetView;
	Microsoft::WRL::ComPtr<ID3D11DepthStencilView> m_depthStencilView;

	// Cached renderer properties.
	D3D_FEATURE_LEVEL m_featureLevel;
	Windows::Foundation::Size m_renderTargetSize;
	Windows::Foundation::Rect m_windowBounds;
	Platform::Agile<Windows::UI::Core::CoreWindow> m_window;
	Windows::Graphics::Display::DisplayOrientations m_orientation;

	// Transform used for display orientation.
	DirectX::XMFLOAT4X4 m_orientationTransform3D;
};
```

If you have read Chapter 4 of the book, you should have no problem following the implementation of this class in the code. The virtual functions and data members should be somewhat reminiscent of those in `D3DApp`. When you derive from `Direct3DBase`, you can override the virtual functions and implement any app specific code. For example, you can override `CreateDeviceResources` to create any resources that depend on the device (such as create textures and buffers), you can override `UpdateForWindowSizeChanged` to update your projection matrix, and you override `Render` to write your drawing code (observe that `Render` is abstract so it must be overridden).

## §2 Constant Buffers

In *Introduction to 3D Game Programming with DirectX 11*, we relied on the Effects framework to manage constant buffers. We merely set values via `ID3DX11EffectVariable`s, and the Effects framework would update the dirty constant buffers when `ID3DX11EffectPass::Apply` was called. In this section, we

show how to manually manage constant buffers, which is necessary in Metro apps where we cannot use the Effects framework.

### §2.1.1 Creation and Updating

As far as the GPU is concerned, a constant buffer is a chunk of memory that will be bound to constant buffer memory. A constant buffer can store a maximum of 4096 `float4` vectors. A constant buffer is represented by the `ID3D11Buffer` type and is created with the `ID3D11Device::CreateBuffer` method just like vertex and index buffers. The constant buffer must be created with the `D3D11_BIND_CONSTANT_BUFFER` flag. Constant buffers are usually always updated at runtime (e.g., the world matrix is updated to move an object, the camera is moving, and etc.); therefore, it is common, and even recommended (see [1]), for constant buffers to have `D3D11_USAGE_DYNAMIC` and `D3D11_CPU_ACCESS_WRITE` set.

The recommended way [1] to update a constant buffer is using the `ID3D11DeviceContext::Map` API with the `D3D11_MAP_WRITE_DISCARD` flag. When a constant buffer is updated, you must update all of its memory. Moreover, you should try to minimize updates for performance reasons. To help minimize updates, it is recommended to organize constant buffer data based on update frequency.

To facilitate updates, and also to provide a backend for CPU read access, it is useful to associate a C++ data structure that mirrors your constant buffer. Then updating the constant buffer essentially amounts to a memory copy operation, and the CPU can read from the CPU backend if necessary. To facilitate this, we adapt the template class from MJP's SampleFramework11 code available at [2]:

```
template<typename T>
class ConstantBuffer
{
public:
	ConstantBuffer() :
		mInitialized(false),
		mBuffer(0)
	{
	}

	~ConstantBuffer()
	{
		ReleaseCOM(mBuffer);
	}

	///
	/// Public structure instance mirroring the data stored in
	/// the constant buffer.
	///
	T Data;

	ID3D11Buffer* Buffer()const
	{
		return mBuffer;
	}

	HRESULT Initialize(ID3D11Device* device)
	{
		HRESULT hr = S_OK;
```

```
            // Make constant buffer multiple of 16 bytes.
            D3D11_BUFFER_DESC desc;
            desc.Usage = D3D11_USAGE_DYNAMIC;
            desc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
            desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
            desc.MiscFlags = 0;
            desc.ByteWidth = static_cast<UINT>(sizeof(T) +
                  (16 - (sizeof(T) % 16)));
            desc.StructureByteStride = 0;

            VR(device->CreateBuffer(&desc, 0, &mBuffer));

            mInitialized = true;

            return hr;
      }

      ///
      /// Copies the system memory constant buffer data to the GPU
      /// constant buffer.  This call should be made as infrequently
      /// as possible.
      ///
      void ApplyChanges(ID3D11DeviceContext* dc)
      {
            Assert(mInitialized, L"ConstantBuffer not initialized.");

            D3D11_MAPPED_SUBRESOURCE mappedResource;
            dc->Map(mBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
            CopyMemory(mappedResource.pData, &Data, sizeof(T));
            dc->Unmap(mBuffer, 0);
      }

private:
      ConstantBuffer(const ConstantBuffer<T>& rhs);
      ConstantBuffer<T>& operator=(const ConstantBuffer<T>& rhs);

private:

      ID3D11Buffer* mBuffer;
      bool mInitialized;
};
```

### §2.1.2 Binding to the Pipeline

Constant buffers can be bound to the different shader stages of the rendering pipeline:

```
// Vertex Shader
void ID3D11DeviceContext::VSSetConstantBuffers(
    UINT StartSlot,
    UINT NumBuffers,
    ID3D11Buffer *const *ppConstantBuffers
);

// Pixel Shader
void ID3D11DeviceContext::PSSetConstantBuffers(
    UINT StartSlot,
    UINT NumBuffers,
    ID3D11Buffer *const *ppConstantBuffers
);
```

All the function signatures are the same:

1. `StartSlot`: The input slot in which to start binding constant buffers. The maximum number of constant buffer slots is defined in the Direct3D header files to be 14:

   ```
   #define      D3D11_COMMONSHADER_CONSTANT_BUFFER_API_SLOT_COUNT     ( 14 )
   ```

   The slots indexed from 0-13.
2. `NumBuffers`: The number of constant buffers we are binding to the constant buffer slots. If the start slot has index $k$ and we are binding $n$ buffers, then we are binding to slots $C_k, C_{k+1}, \ldots, C_{k+n-1}$.
3. `ppConstantBuffers`: Pointer to the first element of an array of constant buffers.

Analogous `*SetConstantBuffers` methods exist for the other shader types.

As an example, in our HLSL code, we define two constant buffers like so:

```
cbuffer cbPerFrame : register(b0)
{
	float4x4 gView;
	float4x4 gInvView;
	float4x4 gProj;
	float4x4 gInvProj;
	float4x4 gViewProj;

	DirectionalLight gDirLights[3];
	float3 gEyePosW;
	float perFramePad0;

	float4 gFogColor;
	float  gFogStart;
	float  gFogRange;
};

cbuffer cbPerObject : register(b1)
{
	float4x4 gWorld;
	float4x4 gWorldInvTranspose;
	float4x4 gTexTransform;
	Material gMaterial;
};
```

The ": register(b0)" indicates that the constant buffer corresponds to slot 0 and ": register(b1)" indicates that the constant buffer corresponds to slot 1. So whatever constant buffer we bind to slot 0 that is the buffer "cbPerFrame" refers to. Likewise, whatever constant buffer we bind to slot 1 that is the buffer "cbPerObject" refers to. So it is up to the application to make sure the right resources are getting bound to the right slots.

### §2.1.3 Constant Buffer Usage Method 1

Let us review how constant buffers were used in *to 3D Game Programming with DirectX 11*. Roughly speaking, each effect had a per-frame and per-object constant buffer. In particular, because each effect has its own per-frame constant buffer, we end up updating the per-frame constant buffer for each effect file. Normally, we do not have many effect files, so this is not much of a performance worry, but we could do better by having one per-frame constant buffer that all shaders share. So that we can get away with only one per-frame constant buffer, we can add all the data we need to cover all shaders, even if some of the shaders do not need all the data; it is not a big deal because we only ever update it once per frame. So the only cost is a bind cost. It is common to put this constant buffer in a "header" file and include it into all of your shader files so that you only have to change one shader file if you need to modify the per-frame constant buffer during your project. You may also wish to always bind this constant buffer to slot 0.

As far as rendering objects, we defined each effect file to have one per-object constant buffer. So for each object we draw with the given effect, we must update the per-object constant buffer before drawing the object. Because we update the constant buffer with the `D3D11_MAP_WRITE_DISCARD` flag, this should not cause a GPU stall. However, because the draw call depends on the contents of the constant buffer, there is a chance of a GPU stall while the GPU waits for the buffer to be updated. For example, it cannot draw the *n*th object until the constant buffer has been updated with the *n*th object's information. Most likely this will not impact performance, as other work such as changing resources, shader programs and setting render states needs to be done also before a draw call is executed. The following code updates constant buffers in a way analogous to how we did it with the Effects framework:

```
//
// Update the per-frame constant buffer.
//
mCam.UpdateViewMatrix();

XMMATRIX view = mCam.View();
XMMATRIX proj = mCam.Proj();
XMMATRIX viewProj = mCam.ViewProj();

XMStoreFloat4x4(&mPerFrameCB.Data.View, XMMatrixTranspose(view));
XMStoreFloat4x4(&mPerFrameCB.Data.InvView,
XMMatrixTranspose(XMMatrixInverse(&XMMatrixDeterminant(view), view)));
XMStoreFloat4x4(&mPerFrameCB.Data.Proj, XMMatrixTranspose(proj));
XMStoreFloat4x4(&mPerFrameCB.Data.InvProj,
XMMatrixTranspose(XMMatrixInverse(&XMMatrixDeterminant(proj), proj)));
XMStoreFloat4x4(&mPerFrameCB.Data.ViewProj, XMMatrixTranspose(viewProj));

mPerFrameCB.Data.DirLights[0] = mDirLights[0];
mPerFrameCB.Data.DirLights[1] = mDirLights[1];
mPerFrameCB.Data.DirLights[2] = mDirLights[2];
mPerFrameCB.Data.EyePosW = mCam.GetPosition();

mPerFrameCB.Data.gFogStart = 10.0f;
mPerFrameCB.Data.gFogRange = 60.0f;
mPerFrameCB.Data.gFogColor = XMFLOAT4(0.65f, 0.65f, 0.65f, 1.0f);

mPerFrameCB.ApplyChanges(m_d3dContext.Get());
```

```
//
// Setup IA stage.
//
UINT stride = sizeof(VertexBasic32);
UINT offset = 0;

m_d3dContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
m_d3dContext->IASetInputLayout(mInputLayout.Get());
m_d3dContext->IASetVertexBuffers(0, 1,
      mShapesVB.GetAddressOf(), &stride, &offset);
m_d3dContext->IASetIndexBuffer(mShapesIB.Get(), DXGI_FORMAT_R32_UINT, 0);

//
// Bind constant buffers.
//
ID3D11Buffer* cbuffers[2] = {mPerFrameCB.Buffer(), mPerObjectCB.Buffer()};
m_d3dContext->VSSetConstantBuffers(0, 2, cbuffers);
m_d3dContext->PSSetConstantBuffers(0, 2, cbuffers);

//
// Set sampler.
//
m_d3dContext->PSSetSamplers(0, 1, mSamLinear.GetAddressOf());

//
// Bind vertex and pixel shaders.
//
m_d3dContext->VSSetShader(mBasicVS.Get(), 0, 0);
m_d3dContext->PSSetShader(mBasicLight3FogPS.Get(), 0, 0);

//
// Draw the grid.
//
XMMATRIX world = XMLoadFloat4x4(&mGridWorld);
XMMATRIX worldInvTranspose = MathHelper::InverseTranspose(world);
XMMATRIX texTransform = XMMatrixScaling(6.0f, 8.0f, 1.0f);

XMStoreFloat4x4(&mPerObjectCB.Data.World, XMMatrixTranspose(world));
XMStoreFloat4x4(&mPerObjectCB.Data.WorldInvTranspose,
XMMatrixTranspose(worldInvTranspose));
XMStoreFloat4x4(&mPerObjectCB.Data.TexTransform,
XMMatrixTranspose(texTransform));
mPerObjectCB.Data.Mat = mGridMat;
mPerObjectCB.ApplyChanges(m_d3dContext.Get());

m_d3dContext->PSSetShaderResources(0, 1, mFloorTexSRV.GetAddressOf());

m_d3dContext->DrawIndexed(mGridIndexCount, mGridIndexOffset,
mGridVertexOffset);

//
// Draw the box.
//
world = XMLoadFloat4x4(&mBoxWorld);
worldInvTranspose = MathHelper::InverseTranspose(world);
texTransform = XMMatrixIdentity();

XMStoreFloat4x4(&mPerObjectCB.Data.World, XMMatrixTranspose(world));
XMStoreFloat4x4(&mPerObjectCB.Data.WorldInvTranspose,
XMMatrixTranspose(worldInvTranspose));
XMStoreFloat4x4(&mPerObjectCB.Data.TexTransform,
XMMatrixTranspose(texTransform));
mPerObjectCB.Data.Mat = mBoxMat;
```

```
mPerObjectCB.ApplyChanges(m_d3dContext.Get());

m_d3dContext->PSSetShaderResources(0, 1, mStoneTexSRV.GetAddressOf());

m_d3dContext->DrawIndexed(mBoxIndexCount, mBoxIndexOffset, mBoxVertexOffset);

//
// Draw the cylinders.
//
for(int i = 0; i < 10; ++i)
{
	world = XMLoadFloat4x4(&mCylWorld[i]);
	worldInvTranspose = MathHelper::InverseTranspose(world);
	texTransform = XMMatrixIdentity();

	XMStoreFloat4x4(&mPerObjectCB.Data.World, XMMatrixTranspose(world));
	XMStoreFloat4x4(&mPerObjectCB.Data.WorldInvTranspose,
		XMMatrixTranspose(worldInvTranspose));
	XMStoreFloat4x4(&mPerObjectCB.Data.TexTransform,
		XMMatrixTranspose(texTransform));
	mPerObjectCB.Data.Mat = mCylinderMat;
	mPerObjectCB.ApplyChanges(m_d3dContext.Get());

	m_d3dContext->PSSetShaderResources(
		0, 1, mBrickTexSRV.GetAddressOf());

	m_d3dContext->DrawIndexed(mCylinderIndexCount,
		mCylinderIndexOffset, mCylinderVertexOffset);
}

//
// Draw the spheres.
//
for(int i = 0; i < 10; ++i)
{
	world = XMLoadFloat4x4(&mSphereWorld[i]);
	worldInvTranspose = MathHelper::InverseTranspose(world);
	texTransform = XMMatrixIdentity();

	XMStoreFloat4x4(&mPerObjectCB.Data.World, XMMatrixTranspose(world));
	XMStoreFloat4x4(&mPerObjectCB.Data.WorldInvTranspose,
		XMMatrixTranspose(worldInvTranspose));
	XMStoreFloat4x4(&mPerObjectCB.Data.TexTransform,
		XMMatrixTranspose(texTransform));
	mPerObjectCB.Data.Mat = mSphereMat;
	mPerObjectCB.ApplyChanges(m_d3dContext.Get());

	m_d3dContext->PSSetShaderResources(0, 1, mStoneTexSRV.GetAddressOf());

	m_d3dContext->DrawIndexed(mSphereIndexCount,
		mSphereIndexOffset, mSphereVertexOffset);
}
```

**Note:** Observe that for matrix variables, we pass the transpose of the matrix. This was not necessary in the Effects framework because the Effects framework took the transpose automatically for us. The reason we pass the transpose is to convert our matrix to column-major format, which orders the matrix elements column-by-column in memory as opposed to row-by-row. By default, matrices are expected to follow a column-major convention in constant buffers, but we work with a row-major convention on the C++ side. Below are example data structures for row- and column-major matrices:

```
struct RowMajor
{
	XMVECTOR Row1;
	XMVECTOR Row2;
	XMVECTOR Row3;
	XMVECTOR Row4;
};

struct ColumnMajor
{
	XMVECTOR Col1;
	XMVECTOR Col2;
	XMVECTOR Col3;
	XMVECTOR Col4;
};
```

Say, for example, we do the following:

```
RowMajor A = {...};
ColumnMajor B;
CopyMemory(&B, &A, sizeof(ColumnMajor));
// Bad, B represents the transpose of A
```

Of course, purely memory-wise the matrix A and B are the same, but due to the difference in how the elements are interpreted (by the row- or column-major convention), the matrix B now represents the transpose of A. That is, the *i*th row of A got copied into the *i*th column of B. By taking the transpose of the matrix A first, we are reordering the data elements to make it column-major (elements stored column-by-column) such that it can be copied directly into a column-major matrix:

```
RowMajor A = {...};
RowMajor T = transpose(A); // Make it column major
ColumnMajor B;
CopyMemory(&B, &T, sizeof(ColumnMajor));
// Good, B represents the same transform as A
```

### §2.1.4 Constant Buffer Usage Method 2

Instead of having one per-object constant buffer per shader, an alternative strategy for constant buffers is as follows:

1. Give each model instance a per-object constant buffer to store per-object values, like its world matrix. This buffer only needs to be updated if its data changes (e.g., its world matrix changes), otherwise it is just set with the bind call.

2. Give each material its own constant buffer to store material values. Examples of material values would be diffuse and specular values, or height scale for a displacement mapped material. Unless you have an in-game material editor, material constant values should not change that often once they are initialized. Moreover, materials can be shared across meshes, so in practice, it makes sense of them to be separate from per-object properties.

This strategy, of course, trades updates for increased memory. However, if the constant buffers are not that big, storing an extra constant buffer per model instance, and a constant buffer per material is a small amount of memory compared to texture memory and vertex/index buffer memory. Moreover, these constant buffers can be reused across rendering passes. For example, for a given object, the per-object constant buffer needed to draw the object for the shadow map pass, the normal/depth buffer generation pass for screen space ambient occlusion, and the main rendering pass is the same. If we were to use only one per-object constant buffer, we would have to update it for each object for every rendering pass in the given frame. For complicated character animations that have many bone matrices, this amount of updating could get expensive.

## §3 Texture Loading

The DirectX team has provided texture loading libraries that work with Metro styled applications, which is available here:

http://blogs.msdn.com/b/chuckw/archive/2011/10/28/directxtex.aspx

The three of interest are:

| Library | Description |
|---|---|
| DDSTextureLoader | Light-weight library to load .DDS files, which is the preferred run-time format for shipping applications. This library does not support resizing or format conversion. You just need to add DDSTextureLoader.h/.cpp to your project. |
| WICTextureLoader | Texture loading functions that support BMP, JPEG, PNG, HD Photo, or other WIC supported file container formats. Supports resizing and DXGI_FORMAT conversion. Can also generate mip-map levels. You just need to add DDSTextureLoader.h/.cpp to your project. You just need to add WICTextureLoader.h/.cpp to your project. |
| DirectXTex | More full-fledged texture library that supports all of the above, as well as writing function and other image processing functions like normal map generation. This library is recommended for tools for the art pipeline rather than run-time use in a game. |

We use DDSTextureLoader in our tutorial sample. We load the textures using the `CreateDDSTextureFromFile` function:

```
HRESULT CreateDDSTextureFromFile(
      _In_ ID3D11Device* d3dDevice,
       _In_z_ const wchar_t* szFileName,
       _Out_opt_ ID3D11Resource** texture,
       _Out_opt_ ID3D11ShaderResourceView** textureView,
```

```
        _In_ size_t maxsize = 0);
```

Below is code from the sample demonstrating its usage:

```
#include "DDSTextureLoader.h"

...

ID3D11Resource* tex = 0;

Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> mFloorTexSRV;
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> mStoneTexSRV;
Microsoft::WRL::ComPtr<ID3D11ShaderResourceView> mBrickTexSRV;

ThrowIfFailed(CreateDDSTextureFromFile(m_d3dDevice.Get(),
      L"Media/Textures/floor.dds", &tex, &mFloorTexSRV));

// SRV keeps reference.
ReleaseCOM(tex);

ThrowIfFailed(CreateDDSTextureFromFile(m_d3dDevice.Get(),
      L"Media/Textures/stone.dds", &tex, &mStoneTexSRV));

// SRV keeps reference.
ReleaseCOM(tex);

ThrowIfFailed(CreateDDSTextureFromFile(m_d3dDevice.Get(),
      L"Media/Textures/bricks.dds", &tex, &mBrickTexSRV));

// SRV keeps reference.
ReleaseCOM(tex);
```

**Note:** Texture files should be added to your Visual Studio project so that they are automatically put into your .appx package. For security, Windows 8 does not allow you to access arbitrary folders. If you do not add your texture files to Visual Studio, then you need to manually put them in your AppX folder, which is located at *Debug\ProjectName\AppX* or *Release\ProjectName\AppX*. The same is true about any other content such as compiled shader programs.

## §4 Creating and Binding Sampler States

In *Introduction to 3D Game Programming with DirectX 11*, we defined our sampler-state objects ("samplers") in our effects code and the effects framework would automatically created the sampler resource and bind it to the pipeline as needed when `ID3DX11EffectPass::Apply` was called:

```
// Defining SamplerState in effect file.
SamplerState samLinear
{
      Filter = MIN_MAG_MIP_LINEAR;
      AddressU = WRAP;
```

```
    AddressV = WRAP;
};
```

Without effect files, we need to create a sampler ourselves. This is done by filling out a `D3D11_SAMPLER_DESC` instance and calling `ID3D11Device::CreateSamplerState`:

```
Microsoft::WRL::ComPtr<ID3D11SamplerState> mSamLinear;

D3D11_SAMPLER_DESC samDesc;
samDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
samDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
samDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
samDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
samDesc.MipLODBias = 0;
samDesc.MaxAnisotropy = 1;
samDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;
samDesc.BorderColor[ 0 ] = 1.0f;
samDesc.BorderColor[ 1 ] = 1.0f;
samDesc.BorderColor[ 2 ] = 1.0f;
samDesc.BorderColor[ 3 ] = 1.0f;
samDesc.MinLOD = -3.402823466e+38F; // -FLT_MAX
samDesc.MaxLOD = 3.402823466e+38F; // FLT_MAX

ThrowIfFailed(m_d3dDevice->CreateSamplerState(&samDesc, &mSamLinear));
```

Different shaders will use different samplers, so Direct3D 11 provides an API to set the samplers separately for each of the different shader stages:

```
void ID3D11DeviceContext::VSSetSamplers(
  [in]  UINT StartSlot,
  [in]  UINT NumSamplers,
  [in]  ID3D11SamplerState *const *ppSamplers
);

void ID3D11DeviceContext::PSSetSamplers(
  [in]  UINT StartSlot,
  [in]  UINT NumSamplers,
  [in]  ID3D11SamplerState *const *ppSamplers
);
```

All the function signatures are the same:

1. `StartSlot`: The input slot in which to start binding samplers. The maximum number of sampler slots is defined in the Direct3D header files to be 16:

   ```
   #define     D3D11_COMMONSHADER_SAMPLER_SLOT_COUNT     ( 16 )
   ```

   The slots indexed from 0-15.
2. `NumSamplers`: The number of samplers we are binding to the sampler slots. If the start slot has index $k$ and we are binding $n$ samplers, then we are binding to slots $C_k, C_{k+1}, \dots, C_{k+n-1}$.

3. `ppConstantBuffers`: Pointer to the first element of an array of samplers.

Analogous `*SetSamplers` methods exist for the other shader types.

Example call:

```
Microsoft::WRL::ComPtr<ID3D11SamplerState> mSamLinear;

...

// We only set to PS in our demo because the VS does not
// need a sampler.
m_d3dContext->PSSetSamplers(0, 1, mSamLinear.GetAddressOf());
```

In our HLSL code, we declare the sampler like so:

```
SamplerState samLinear : register(s0);
```

The ": register(s0)" indicates that this sampler corresponds to slot 0. So whatever sampler we bind to slot 0 that is the sampler "samLinear" refers to. So it is up to the application to make sure the right resources are getting bound to the right slots.

## §5 Binding Shader Resource Views

We also relied on the effects framework for binding our shader resource views (SRVs). We already know how to create SRVs, so we just need to learn how to bind them to the pipeline ourselves. Different shaders will use different SRVs, so Direct3D 11 provides an API to set the SRVs separately for each of the different shader stages:

```
void ID3D11DeviceContext::VSSetShaderResources(
  [in]  UINT StartSlot,
  [in]  UINT NumViews,
  [in]  ID3D11ShaderResourceView *const *ppShaderResourceViews
);

void ID3D11DeviceContext::PSSetShaderResources(
  [in]  UINT StartSlot,
  [in]  UINT NumViews,
  [in]  ID3D11ShaderResourceView *const *ppShaderResourceViews
);
```

All the function signatures are the same:

1. `StartSlot`: The input slot in which to start binding SRVs. The maximum number of sampler slots is defined in the Direct3D header files to be 128:

   ```
   #define      D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT   ( 128 )
   ```

   The slots indexed from 0-127.

2. `NumSamplers`: The number of SRVs we are binding to the SRV slots. If the start slot has index $k$ and we are binding $n$ SRVs, then we are binding to slots $C_k, C_{k+1}, \ldots, C_{k+n-1}$.
3. `ppConstantBuffers`: Pointer to the first element of an array of SRVs.

Analogous `*SetShaderResources` methods exist for the other shader types.

Example call:

```
m_d3dContext->PSSetShaderResources(0, 1, mFloorTexSRV.GetAddressOf());
```

In our HLSL code, we declare the SRV (to a texture) like so:

```
Texture2D gDiffuseMap : register(t0);
```

The ": register(t0)" indicates that this SRV corresponds to slot 0. So whatever SRV we bind to slot 0 that is the SRV "gDiffuseMap" refers to. So it is up to the application to make sure the right resources are getting bound to the right slots.

## §6 Using Shaders without the Effects Framework

When we write shader programs without the effects framework, it is common to use a separate file per shader program; for example, if you had a vertex and pixel shader, you would write the vertex shader in one file and the pixel shader in another file. Additionally, it is common to use the extension .hlsl for (high-level-shader-language). Below are the vertex and pixel shaders we use in our demo:

**BasicBaseVS.hlsl**

```
// Include our common lighting code, as well as per-frame constant buffer.
#include "../ShaderInclude.hlsl"

cbuffer cbPerObject : register(b1)
{
	float4x4 gWorld;
	float4x4 gWorldInvTranspose;
	float4x4 gTexTransform;
	Material gMaterial;
};


struct VertexIn
{
	float3 PosL    : POSITION;
	float3 NormalL : NORMAL;
	float2 Tex     : TEXCOORD;
};

struct VertexOut
{
	float3 PosW    : POSITION;
	float3 NormalW : NORMAL;
	float2 Tex     : TEXCOORD;
	float4 PosH    : SV_POSITION;
};
```

```
VertexOut main(VertexIn vin)
{
	VertexOut vout;

	// Transform to world space space.
	vout.PosW    = mul(float4(vin.PosL, 1.0f), gWorld).xyz;
	vout.NormalW = mul(vin.NormalL, (float3x3)gWorldInvTranspose);

	// Transform to homogeneous clip space.
	vout.PosH = mul(float4(vout.PosW, 1.0f), gViewProj);

	// Output vertex attributes for interpolation across triangle.
	vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;

	return vout;
}
```

**BasicBasePS.hlsl**

```
// Include our common lighting code, as well as per-frame constant buffer.
#include "../ShaderInclude.hlsl"

#ifndef ALPHA_CLIP
#define ALPHA_CLIP 0
#endif

#ifndef LIGHT_COUNT
#define LIGHT_COUNT 0
#endif

#ifndef FOG_ENABLED
#define FOG_ENABLED 0
#endif


cbuffer cbPerObject : register(b1)
{
	float4x4 gWorld;
	float4x4 gWorldInvTranspose;
	float4x4 gTexTransform;
	Material gMaterial;
};

Texture2D gDiffuseMap : register(t0);
SamplerState samLinear : register(s0);

struct PixelIn
{
	float3 PosW    : POSITION;
	float3 NormalW : NORMAL;
	float2 Tex     : TEXCOORD;
};

float4 main(PixelIn pin) : SV_Target
{
	// Interpolating normal can unnormalize it, so normalize it.
	pin.NormalW = normalize(pin.NormalW);

	// The toEye vector is used in lighting.
	float3 toEye = gEyePosW - pin.PosW;
```

```
	// Cache the distance to the eye from this surface point.
	float distToEye = length(toEye);

	// Normalize.
	toEye /= distToEye;

	// Sample texture.
	float4 texColor = gDiffuseMap.Sample( samLinear, pin.Tex );

#if ALPHA_CLIP==1
	// Discard pixel if texture alpha < 0.1.  Note that we do this
	// test as soon as possible so that we can potentially exit the shader
	// early, thereby skipping the rest of the shader code.
	clip(texColor.a - 0.1f);
#endif

	//
	// Lighting.
	//

	float4 litColor = texColor;

#if LIGHT_COUNT>0
	// Start with a sum of zero.
	float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);
	float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);
	float4 spec    = float4(0.0f, 0.0f, 0.0f, 0.0f);

	// Sum the light contribution from each light source.
	[unroll]
	for(int i = 0; i < LIGHT_COUNT; ++i)
	{
		float4 A, D, S;
		ComputeDirectionalLight(gMaterial, gDirLights[i],
		   pin.NormalW, toEye, A, D, S);

		ambient += A;
		diffuse += D;
		spec    += S;
	}

	// Modulate with late add.
	litColor = texColor*(ambient + diffuse) + spec;
#endif

	//
	// Fogging
	//

#if FOG_ENABLED==1
	float fogLerp = saturate( (distToEye - gFogStart) / gFogRange );

	// Blend the fog color and the lit color.
	litColor = lerp(litColor, gFogColor, fogLerp);
#endif

	// Common to take alpha from diffuse material and texture.
	litColor.a = gMaterial.Diffuse.a * texColor.a;

	return litColor;
}
```

## §6.1 Compiling Shaders

In Metro applications, you cannot link with D3DCompiler.lib. Metro only allows certain APIs to be used for security (this is also why D3DX library and Effects cannot be used). Therefore, it is necessary to compile your shaders offline using FXC and ship your compiled shader code in the .appx package. You should either add your compiled shader programs to Visual Studio project so that they are automatically put into your .appx package, or manually put them into your AppX folder.

Because we are not using the Effects framework, we cannot define various techniques with uniform parameters to generate different shader variations. However, we can achieve the same end by using conditional compilation. Below are the FXC calls to compile the above vertex and pixel shader; we use conditional compilation to generate a pixel shader that uses 3 lights with fog enabled.

```
fxc /Fc /Od /Zi /T vs_5_0  /E main /Fo BasicBaseVS.cso BasicVS.hlsl
fxc /Fc /Od /Zi /T ps_5_0  /E main /D LIGHT_COUNT=3 /D FOG_ENABLED=1
/Fo BasicPS_light3Fog.cso BasicBasePS.hlsl
```

Observed that the compiled shader object code file has extension .cso.

Now, you do not actually need to fire up FXC yourself. VS12 recognizes .hlsl files and will invoke FXC on them as part of your build process. To see this, add the .hlsl file to your project, right click on it in Solution Explorer, and select Properties. The dialog box as shown in Figure 2 appears. Now, instead of passing command line arguments, you have a nice user interface to set the different compilation options. Moreover, by using VS12, your compiled shader code will automatically be added to your AppX folder.

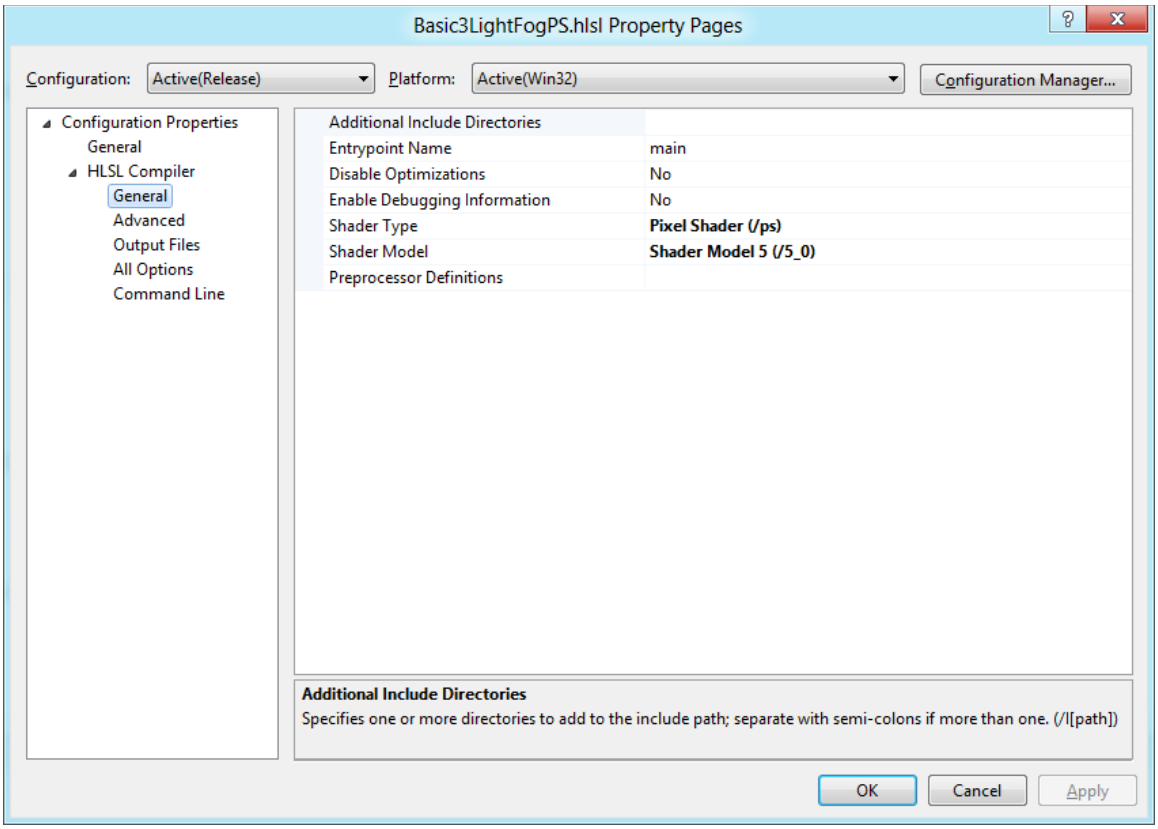


Figure 2

Using VS12 presents a minor problem when using conditional compilation. You cannot have more than one compiled shader output file per .hlsl file. However, that is exactly what we want to do—compile the same .hlsl multiple times with different macro definitions to generate different shaders. A simple workaround that is practical for small projects is to create a separate .hlsl for each shader variation that does nothing more than #include the real shader code and define the appropriate macros. For example, to generate a "basic" pixel shader with 3 lights and fog enabled would be done by creating the following *Basic3LightFogPS.hlsl* file, adding it to VS12, and setting the appropriate HLSL Compiler properties.

**Basic3LightFogPS.hlsl**

```
// Define macros to customize shader.
#define ALPHA_CLIP 0
#define LIGHT_COUNT 3
#define FOG_ENABLED 1

// Include the base shader code.
#include "BasicBasePS.hlsl"
```

## §6.2 Creating and Binding Shaders

The Direct3D project template provides a helper function `ReadDataAsync` (defined in *DirectXHelper.h*) to asynchronously load data using the Parallel Patterns Library Tasks API. Briefly, the Tasks API allows us to chain together operations that must happen

sequentially, but where the entire chain is executed asynchronously. For example, we can load the compiled vertex shader data from file and after create a vertex shader in a background thread while the main thread continues to feel responsive.

```
Microsoft::WRL::ComPtr<ID3D11VertexShader> mBasicVS;

// Load the compiled shader data from file asynchronously.
auto loadVSTask = ReadDataAsync("BasicBaseVS.cso");

// Then, after its loaded...
auto createVSTask = loadVSTask.then([this](Platform::Array<byte>^
fileData)
{
      // Create the vertex shader
      ThrowIfFailed(m_d3dDevice->CreateVertexShader(
            fileData->Data, fileData->Length,
            nullptr, &mBasicVS));

      // Other work
```

As shown above, we create a vertex shader with the following method:

```
HRESULT ID3D11Device::CreateVertexShader(
    const void *pShaderBytecode,
    SIZE_T BytecodeLength,
    ID3D11ClassLinkage *pClassLinkage,
    ID3D11VertexShader **ppVertexShader);
```

1. `pShaderBytecode`: Pointer to the compiled shader data.
2. `BytecodeLength`: The size, in bytes, of the compiled shader data.
3. `pClassLinkage`: Used for HLSL dynamic linkage—this is not covered in the book.
4. `ppVertexShader`: Returns the created vertex shader.

Similarly, a pixel shader is created with this method:

```
HRESULT  ID3D11Device::CreatePixelShader(
     const void *pShaderBytecode,
     SIZE_T BytecodeLength,
     ID3D11ClassLinkage *pClassLinkage,
     ID3D11PixelShader **ppPixelShader);
```

Analogous `Create*Shader` methods exist for the other kinds of Direct3D 11 shaders.

To use a created shader, it must be bound to the rendering pipeline. As with all objects we bind to the pipeline, they stay in effect until we replace them by binding a different object. Vertex and pixel shaders are bound with the following methods:

```
void ID3D11DeviceContext::VSSetShader(
  [in]  ID3D11VertexShader *pVertexShader,
  [in]  ID3D11ClassInstance *const *ppClassInstances,
  [in]  UINT NumClassInstances
);
```

```
void ID3D11DeviceContext::PSSetShader(
  [in]  ID3D11PixelShader *pPixelShader,
  [in]  ID3D11ClassInstance *const *ppClassInstances,
  [in]  UINT NumClassInstances
);
```

1. `pShader`: The shader to bind.
2. `ppClassInstances`: Used for HLSL dynamic linkage—this is not covered in the book.
3. `NumClassInstances`: Used for HLSL dynamic linkage—this is not covered in the book.

Analogous `*SetShader` methods exist for the other kinds of Direct3D 11 shaders.

Below is an example of how we bind our vertex and pixel shaders to the pipeline:

```
Microsoft::WRL::ComPtr<ID3D11VertexShader> mBasicVS;
Microsoft::WRL::ComPtr<ID3D11PixelShader> mBasicLight3FogPS;

...

m_d3dContext->VSSetShader(mBasicVS.Get(), 0, 0);
m_d3dContext->PSSetShader(mBasicLight3FogPS.Get(), 0, 0);
```

## §7 Demo Controls

Mouse control is handled by attaching handlers to the following events:

- CoreWindow.PointerPressed
- CoreWindow.PointerMoved
- CoreWindow.PointerReleased

```
void MetroRenderer::CreateDeviceResources()
{
    Direct3DBase::CreateDeviceResources();

    m_window->PointerPressed +=
        ref new TypedEventHandler<CoreWindow^, PointerEventArgs^>(
        this, &MetroRenderer::OnPointerPressed);
    m_window->PointerMoved   +=
        ref new TypedEventHandler<CoreWindow^, PointerEventArgs^>(
        this, &MetroRenderer::OnPointerMoved);
    m_window->PointerReleased +=
        ref new TypedEventHandler<CoreWindow^, PointerEventArgs^>(
        this, &MetroRenderer::OnPointerReleased);

...
}

void MetroRenderer::OnPointerPressed(CoreWindow^ sender, PointerEventArgs^ args)
{
        mLastMousePos = args->CurrentPoint->Position;

        m_window->SetPointerCapture();

        mMouseDown = true;
```

```
}

void MetroRenderer::OnPointerMoved(CoreWindow^ sender, PointerEventArgs^ args)
{
	Point currPoint = args->CurrentPoint->Position;

	if(args->CurrentPoint->Properties->IsLeftButtonPressed)
	{
		// Make each pixel correspond to a quarter of a degree.
		float dx = XMConvertToRadians(
			0.25f*static_cast<float>(currPoint.X - mLastMousePos.X));
		float dy = XMConvertToRadians(
			0.25f*static_cast<float>(currPoint.Y - mLastMousePos.Y));

		mCam.Pitch(dy);
		mCam.RotateY(dx);
	}
	else if(args->CurrentPoint->Properties->IsRightButtonPressed)
	{
		float dx = 0.02f*static_cast<float>(currPoint.X - mLastMousePos.X);
		float dy = 0.02f*static_cast<float>(currPoint.Y - mLastMousePos.Y);

		mCam.Walk(dy);
		mCam.Strafe(dx);
	}

	mLastMousePos = currPoint;
}

void MetroRenderer::OnPointerReleased(CoreWindow^ sender, PointerEventArgs^ args)
{
	mMouseDown = false;

	m_window->ReleasePointerCapture();
}
```

Our demo uses the following controls:

- Hold the left mouse button down and move the mouse to look around.
- Hold the right mouse button down and move the mouse to move the camera.

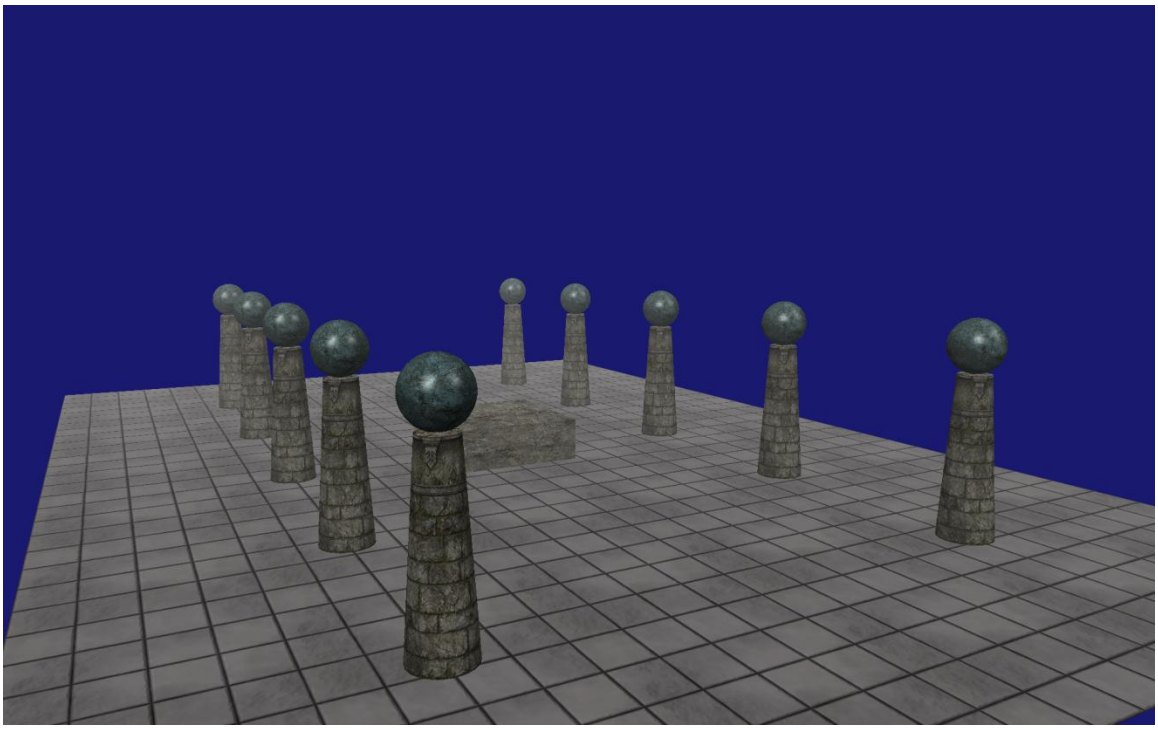

Figure 3: Screenshot of demo.

## §8 Summary

1. A constant buffer is represented by the `ID3D11Buffer` type and is created with the `ID3D11Device::CreateBuffer` method. The constant buffer must be created with the `D3D11_BIND_CONSTANT_BUFFER` flag. For constant buffers that are updated frequently, specify the `D3D11_USAGE_DYNAMIC` and `D3D11_CPU_ACCESS_WRITE` creation flags, and use the `ID3D11DeviceContext::Map` API with the `D3D11_MAP_WRITE_DISCARD` flag. Use the various `ID3D11DeviceContext::*SetConstantBuffers` methods to bind a constant buffer resource to a stage of the rendering pipeline.

2. Use the various alternative texture loading libraries like DDSTextureLoader, WICTextureLoader, and DirectXTex, available at http://blogs.msdn.com/b/chuckw/archive/2011/10/28/directxtex.aspx, as a replacement for the D3DX textures loading functions. When not using the Effects framework, we must create our sampler-state objects (`ID3D11SamplerState`) by filling out a `D3D11_SAMPLER_DESC` instance and calling the `ID3D11Device::CreateSamplerState` function. Use the various `ID3D11DeviceContext::*SetSamplers` methods to bind a sampler-state object to a stage of the rendering pipeline. Use the various `ID3D11DeviceContext::*SetShaderResources` methods to bind a SRVs to a stage of the rendering pipeline.

3. For Metro apps, it is necessary to compile your shaders offline using FXC and ship your compiled shader code in the .appx package. You can use conditional compilation and macros to generate different variations of a "base" shader program. Shader objects can be created with the `ID3D11Device::Create*Shader` methods, and shaders can be bound to the stages of the rendering pipeline with the `ID3D11DeviceContext::*SetShader` methods.

## §9 References


[1] John McDonald, NVIDIA "Don't Throw it All Away – Managing Buffers Efficiently" http://developer.nvidia.com/sites/default/files/akamai/gamedev/files/gdc12/Efficient_Buffer_Management_McDonald.pdf

[2] http://mynameismjp.wordpress.com/