Week4

Hooman Salamat

Drawing in Direct3D

This week, you will:

- Discover the Direct3D interfaces methods for defining, storing, and drawing geometric data
- Learn how to write basic vertex and pixel shaders
- Find out how to configure the rendering pipeline with pipeline state objects
- Learn about root descriptors and root constants
- Discover how to procedurally generate and draw common geometric shapes like grids, cylinders, and spheres
- Find out how we can animate vertices on the CPU and upload the new vertex positions to the GPU

Drawing

- We now will focus on the Direct3D API interfaces and methods needed to configure the rendering pipeline, define vertex and pixel shaders, and submit geometry to the rendering pipeline for drawing

- You will soon be able to draw a 3D box with solid coloring or in wireframe mode

- Vertices and Input Layouts

▪ To create a custom vertex format, we first create a structure that holds the vertex data we choose

▪ Here we have two different kinds of vertex formats

▪ One consists of position and color

▪ The second consists of position, normal vector, and two sets of 2D texture coordinates

```
struct Vertex1

{

XMFLOAT3 Pos;

XMFLOAT4 Color;

};

struct Vertex2

{

XMFLOAT3 Pos;

XMFLOAT3 Normal;

XMFLOAT2 Tex0;

XMFLOAT2 Tex1;

};
```

Input layout description

- We need to provide Direct3D with a description of our vertex structure so that it knows what to do with each component

- D3D12_INPUT_LAYOUT_DESC describes the input-buffer data for the input-assembler stage.

- If the vertex structure D3D12_INPUT_ELEMENT_DESC has two components, then the corresponding array will have two elements

https://docs.microsoft.com/en-ca/windows/win32/api/d3d12/ns-d3d12-d3d12_input_element_desc

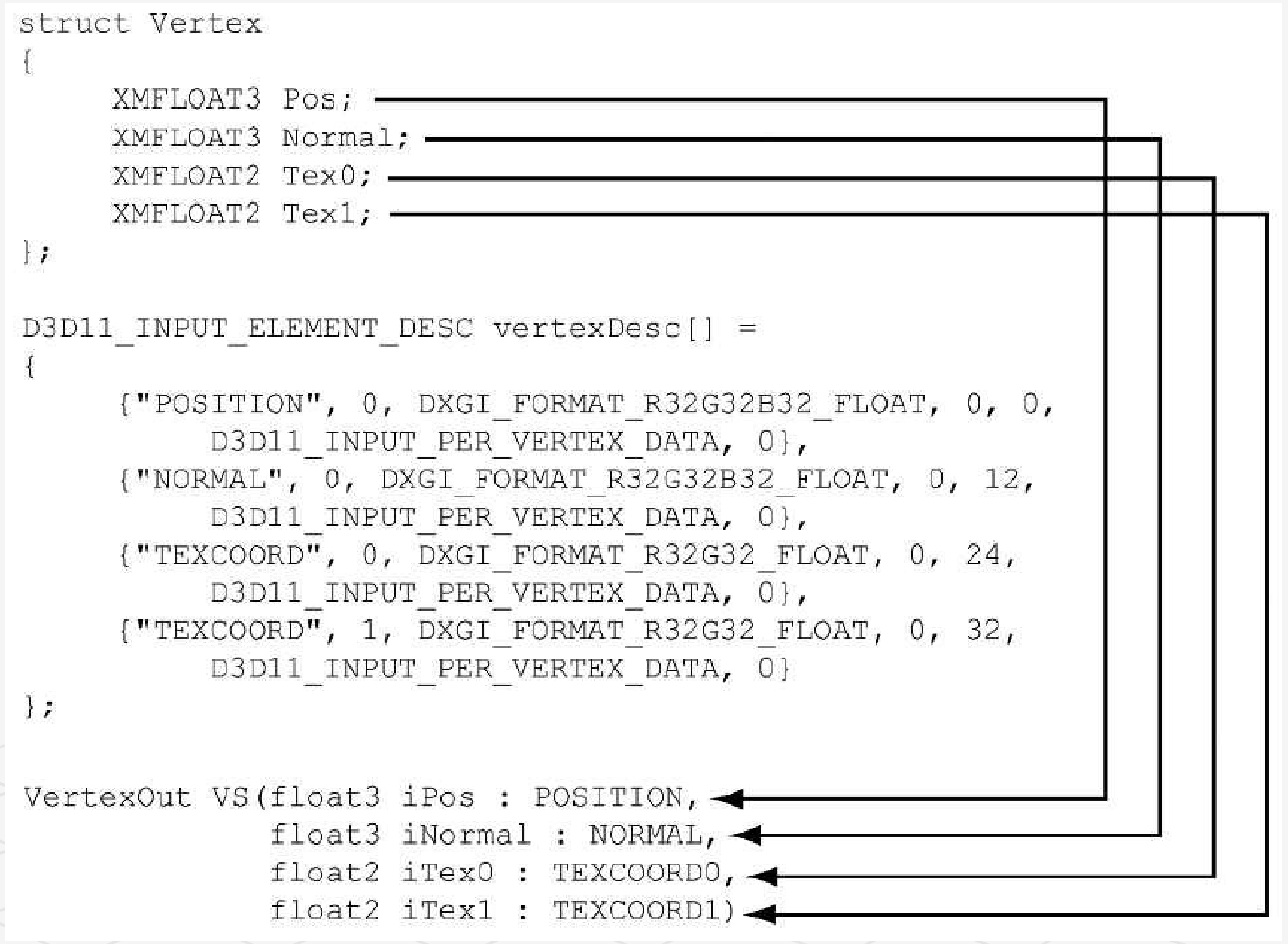
```
typedef struct D3D12_INPUT_LAYOUT_DESC {

const D3D12_INPUT_ELEMENT_DESC *pInputElementDescs;

UINT NumElements; } D3D12_INPUT_LAYOUT_DESC;
```

```
typedef struct D3D12_INPUT_ELEMENT_DESC
    {
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D12_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
    } D3D12_INPUT_ELEMENT_DESC;
```

D3D12_INPUT_ELEMENT_DESC array

Each element in the vertex structure is described by a corresponding element in the D3D12_INPUT_ELEMENT_DESC array.

The semantic name and index provides way for mapping vertex elements to the corresponding parameters of the vertex shader.



```
struct Vertex
{
     XMFLOAT3 Pos;
     XMFLOAT3 Normal;
     XMFLOAT2 Tex0;
     XMFLOAT2 Tex1;
};

D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
     {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
          D3D11_INPUT_PER_VERTEX_DATA, 0},
     {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
          D3D11_INPUT_PER_VERTEX_DATA, 0},
     {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
          D3D11_INPUT_PER_VERTEX_DATA, 0},
     {"TEXCOORD", 1, DXGI_FORMAT_R32G32_FLOAT, 0, 32,
          D3D11_INPUT_PER_VERTEX_DATA, 0}
};

VertexOut VS(float3 iPos : POSITION,
             float3 iNormal : NORMAL,
             float2 iTex0 : TEXCOORD0,
             float2 iTex1 : TEXCOORD1)
```

Textures Formats

- A texture can only store certain kinds of data element formats
- Described by the DXGI_FORMAT enumerated type
- Examples:
- DXGI_FORMAT_R32G32B32_FLOAT
- DXGI_FORMAT_R16G16B16A16_UNORM (0 to 1 range)
- DXGI_FORMAT_R32G32_UINT
- DXGI_FORMAT_R8G8B8A8_UNORM (0 to 1 range)
- DXGI_FORMAT_R8G8B8A8_SNORM (-1 to 1 range)
- DXGI_FORMAT_R8G8B8A8_SINT (-128 to 127 range)
- DXGI_FORMAT_R8G8B8A8_UINT (0 to 255 range)

Offset

5. AlignedByteOffset: The offset, in bytes, from the start of the C++ vertex structure of the specified input slot to the start of the vertex component. For example, in the following vertex structure,

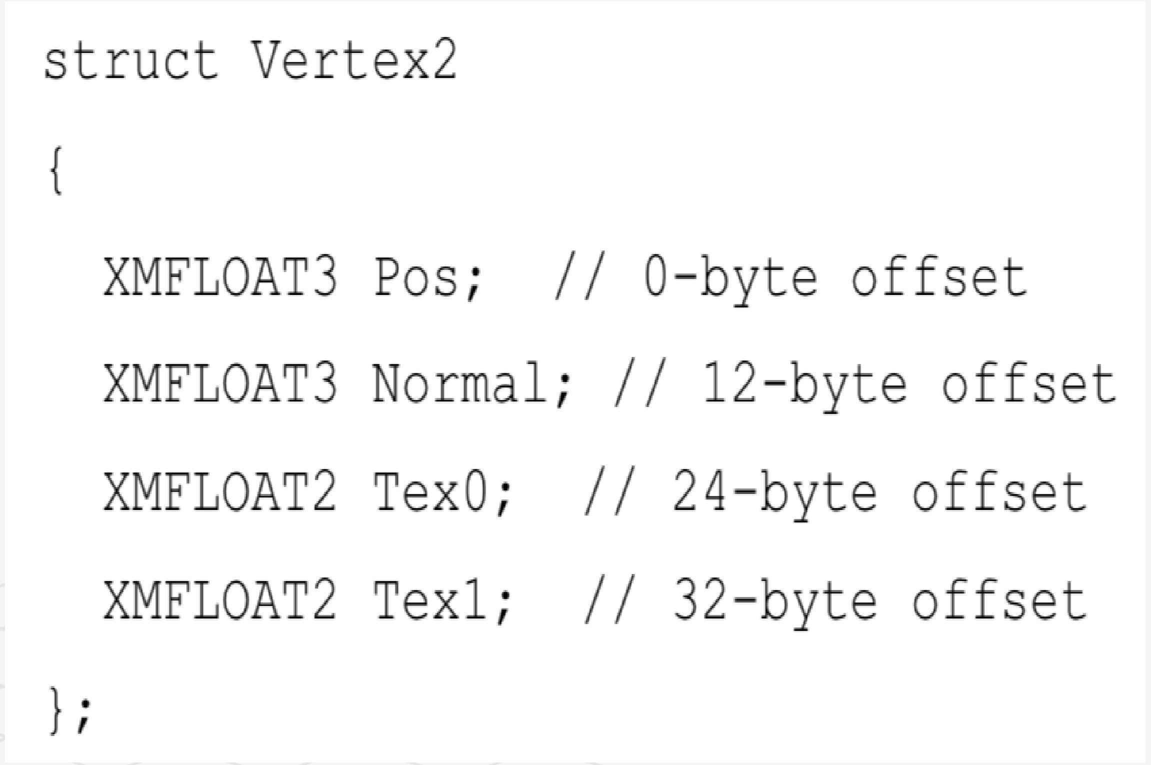
the element Pos has a 0-byte offset since its start coincides with the start of the vertex structure;

the element Normal has a 12-byte offset because we have to skip over the bytes of Pos to get to the start of Normal;

the element Tex0 has a 24-byte offset because we need to skip over the bytes of Pos

Normal to get to the start of Tex0;

the element Tex1 has a 32-byte offset. because we need to skip over the bytes of Pos, Normal, and Tex0 to get to the start of Tex1.



```
struct Vertex2
{
  XMFLOAT3 Pos;  // 0-byte offset
  XMFLOAT3 Normal; // 12-byte offset
  XMFLOAT2 Tex0;  // 24-byte offset
  XMFLOAT2 Tex1;  // 32-byte offset
};
```

D3D12HelloTriangle

```
//D3D12HelloTriangle.h

Struct Vertex

    {

        XMFLOAT3 position;

        XMFLOAT4 color;

    };
```

```
//D3D12HelloTriangle.cpp void D3D12HelloTriangle::LoadAssets()

 // Define the vertex input layout.

        D3D12_INPUT_ELEMENT_DESC inputElementDescs[] =

        {

            { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },

            { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }

        };
```

Box

Vertex type and colors are defined as:

```
struct Vertex

{

    XMFLOAT3 Pos;

    XMFLOAT4 Color;

};

std::vector<D3D12_INPUT_ELEMENT_DESC> mInputLayout;

// void BoxApp::BuildShadersAndInputLayout()

mInputLayout =

    {

        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },

        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }

    };
```

VERTEX BUFFERS

 In order for the GPU to access an array of vertices, they need to be placed in a resource (ID3D12Resource) called a buffer. We call a buffer that stores vertices a *vertex buffer*. Buffers are simpler resources than textures; they are not multidimensional, and do not have mipmaps, filters, or multisampling support. We will use buffers whenever we need to provide the GPU with an array of data elements such as vertices.

 Because an intermediate upload buffer is required to initialize the data of a default buffer, we build a utility function in d3dUtil.h/.cpp called CreateDefaultBuffer

The code on the right shows how this class would be used to create a default buffer that stored the 8 vertices of a cube, where each vertex had a different color associated with it

```
void BoxApp::BuildBoxGeometry()

{

    std::array<Vertex, 8> vertices =

    {

Vertex({ XMFLOAT3(-1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::White) }),

Vertex({ XMFLOAT3(-1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Black) }),

Vertex({ XMFLOAT3(+1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Red) }),

Vertex({ XMFLOAT3(+1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::Green) }),

Vertex({ XMFLOAT3(-1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Blue) }),

Vertex({ XMFLOAT3(-1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Yellow) }),

Vertex({ XMFLOAT3(+1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Cyan) }),

Vertex({ XMFLOAT3(+1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Magenta) })

    };
```

ID3D12Resource

we create an ID3D12Resource object by filling out a D3D12_RESOURCE_DESC structure describing the buffer resource, and then calling the ID3D12Device::CreateCommittedResource method

For static geometry (i.e., geometry that does not change on a per-frame basis), we put vertex buffers in the default heap (D3D12_HEAP_TYPE_DEFAULT) for optimal performance.

After the vertex buffer has been initialized, only the GPU needs to read from the vertex buffer to draw the geometry

If the CPU cannot write to the vertex buffer in the default heap, how do we initialize the vertex buffer? we need to create an intermediate *upload* buffer resource with heap type D3D12_HEAP_TYPE_UPLOAD.

Direct3D 12 provides a C++ wrapper class CD3DX12_RESOURCE_DESC, which derives from D3D12_RESOURCE_DESC and provides convenience constructors and methods.

```
Microsoft::WRL::ComPtr<ID3D12Resource> d3dUtil::CreateDefaultBuffer(
    ID3D12Device* device,
    ID3D12GraphicsCommandList* cmdList,
    const void* initData,
    UINT64 byteSize,
    Microsoft::WRL::ComPtr<ID3D12Resource>& uploadBuffer)
{
    ComPtr<ID3D12Resource> defaultBuffer;

    // Create the actual default buffer resource.
    ThrowIfFailed(device->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
        D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(byteSize),
          D3D12_RESOURCE_STATE_COMMON,
        nullptr,
        IID_PPV_ARGS(defaultBuffer.GetAddressOf())));

// In order to copy CPU memory data into our default buffer, we need to
create an intermediate upload heap.
    ThrowIfFailed(device->CreateCommittedResource(
        &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
D3D12_HEAP_FLAG_NONE,
        &CD3DX12_RESOURCE_DESC::Buffer(byteSize),
D3D12_RESOURCE_STATE_GENERIC_READ,
        nullptr,
        IID_PPV_ARGS(uploadBuffer.GetAddressOf())));
```

CD3DX12_RESOURCE_DESC

Describes a resource. Two common resources are buffers and textures, which both use this structure, but with quite different uses of the fields.

```
CD3DX12_RESOURCE_DESC(
        D3D12_RESOURCE_DIMENSION dimension,
        UINT64 alignment,
        UINT64 width,
        UINT height,
        UINT16 depthOrArraySize,
        UINT16 mipLevels,
        DXGI_FORMAT format,
        UINT sampleCount,
        UINT sampleQuality,
        D3D12_TEXTURE_LAYOUT layout,
        D3D12_RESOURCE_FLAGS flags )
    {
        Dimension = dimension;
        Alignment = alignment;
        Width = width;
        Height = height;
        DepthOrArraySize = depthOrArraySize;
        MipLevels = mipLevels;
        Format = format;
        SampleDesc.Count = sampleCount;
        SampleDesc.Quality = sampleQuality;
        Layout = layout;
        Flags = flags;
    }
```

```
static inline CD3DX12_RESOURCE_DESC Buffer(
            const D3D12_RESOURCE_ALLOCATION_INFO& resAllocInfo,
            D3D12_RESOURCE_FLAGS flags = D3D12_RESOURCE_FLAG_NONE )
    {
            return CD3DX12_RESOURCE_DESC( D3D12_RESOURCE_DIMENSION_BUFFER,
resAllocInfo.Alignment, resAllocInfo.SizeInBytes,
                1, 1, 1, DXGI_FORMAT_UNKNOWN, 1, 0, D3D12_TEXTURE_LAYOUT_ROW_MAJOR,
flags );
    }

    static inline CD3DX12_RESOURCE_DESC Buffer(
            UINT64 width,
            D3D12_RESOURCE_FLAGS flags = D3D12_RESOURCE_FLAG_NONE,
            UINT64 alignment = 0 )
    {
            return CD3DX12_RESOURCE_DESC( D3D12_RESOURCE_DIMENSION_BUFFER, alignment,
width, 1, 1, 1,
                DXGI_FORMAT_UNKNOWN, 1, 0, D3D12_TEXTURE_LAYOUT_ROW_MAJOR, flags );
    }
```

Textures

The CD3DX12_RESOURCE_DESC *class also provides convenience methods for constructing a* D3D12_RESOURCE_DESC *that describes texture resources and querying information about the resource:*

Textures are a multi-dimensional arrangement of texels in a contiguous region of memory, heavily optimized to maximize bandwidth for rendering and sampling. Texture sizes are hard to predict and vary from adapter to adapter. Applications must use ID3D12Device::GetResourceAllocationInfo to accurately understand their size.

Width, *Height*, and *DepthOrArraySize* must be between 1 and the maximum dimension supported for the particular feature level and texture dimension.

1. CD3DX12_RESOURCE_DESC::Tex1D

2. CD3DX12_RESOURCE_DESC::Tex2D

3. CD3DX12_RESOURCE_DESC::Tex3D

https://docs.microsoft.com/en-us/windows/win32/api/d3d12/ns-d3d12-d3d12_resource_desc

D3D12_SUBRESOURCE_DATA

Describes subresource data.

typedef struct D3D12_SUBRESOURCE_DATA { const void *pData; LONG_PTR RowPitch; LONG_PTR SlicePitch; } D3D12_SUBRESOURCE_DATA;

1. pData: A pointer to a system memory array which contains the data to initialize the buffer with. If the buffer can store *n* vertices, then the system array must contain at least *n* vertices so that the entire buffer can be initialized.

2. RowPitch: For buffers, the size of the data we are copying in bytes.

3. SlicePitch: For buffers, the size of the data we are copying in bytes.

```
 // Describe the data we want to copy into the default buffer.
    D3D12_SUBRESOURCE_DATA subResourceData = {};
    subResourceData.pData = initData;
    subResourceData.RowPitch = byteSize;
    subResourceData.SlicePitch = subResourceData.RowPitch;

// Schedule to copy the data to the default buffer resource.  At a high level, the
//helper function UpdateSubresources will copy the CPU memory into the intermediate upload
//heap.  Then, using ID3D12CommandList::CopySubresourceRegion,
// the intermediate upload heap data will be copied to mBuffer.
cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(defaultBuffer.Get(),
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_COPY_DEST));
    UpdateSubresources<1>(cmdList, defaultBuffer.Get(), uploadBuffer.Get(), 0, 0, 1,
&subResourceData);
cmdList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(defaultBuffer.Get(),
D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_GENERIC_READ));

    // Note: uploadBuffer has to be kept alive after the above function calls because
    // the command list has not been executed yet that performs the actual copy.
    // The caller can Release the uploadBuffer after it knows the copy has been executed.


    return defaultBuffer;
}
```

Vertex buffer view

In order to bind a vertex buffer to the pipeline, we need to create a vertex buffer view

to the vertex buffer resource. Unlike an RTV (render target view), we do not need a descriptor heap for a vertex buffer view.

A vertex buffer view is represented by the D3D12_VERTEX_BUFFER_VIEW_DESC structure.

After creating the view we need to bind it to an input slot of the pipeline.

```
typedef struct D3D12_VERTEX_BUFFER_VIEW
    {
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
    UINT SizeInBytes;
    UINT StrideInBytes;
    } D3D12_VERTEX_BUFFER_VIEW;
```

```
void IASetVertexBuffers(
UINT StartSlot,
UINT NumViews,
const D3D12_VERTEX_BUFFER_VIEW *pViews)
```

```
D3D12_VERTEX_BUFFER_VIEW vbv;
vbv.BufferLocation = VertexBufferGPU -> GetGPUVirtualAddress();
vbv.StrideInBytes = sizeof(Vertex);
vbv.SizeInBytes = 8 * sizeof(Vertex);
D3D12_VERTEX_BUFFER_VIEW vertexBuffers[1] = { vbv };
mCommandList->IASetVertexBuffers(0, 1, vertexBuffers);
```

DrawInstanced

A vertex buffer will stay bound to an input slot until you change it. So you may structure your code like this, if you are using more than one vertex buffer:

```
ID3D12Resource* mVB1; // stores vertices of type Vertex1
ID3D12Resource* mVB2; // stores vertices of type Vertex2
D3D12_VERTEX_BUFFER_VIEW_DESC mVBView1; // view to mVB1
D3D12_VERTEX_BUFFER_VIEW_DESC mVBView2; // view to mVB2
/*…Create the vertex buffers and views…*/
mCommandList->IASetVertexBuffers(0, 1, &VBView1);
/* …draw objects using vertex buffer 1… */
mCommandList->IASetVertexBuffers(0, 1, &mVBView2);
/* …draw objects using vertex buffer 2… */
```

Setting a vertex buffer to an input slot does not draw them; it only makes the vertices ready to be fed into the pipeline. The final step to actually draw the vertices is done with the ID3D12GraphicsCommandList::DrawInstanced method:

```
void DrawInstanced( UINT VertexCountPerInstance, UINT InstanceCount, UINT StartVertexLocation, UINT StartInstanceLocation );
```

VertexCountPerInstance: Number of vertices to draw.

InstanceCount: Number of instances to draw.

StartVertexLocation: Index of the first vertex.

StartInstanceLocation: A value added to each index before reading per-instance data from a vertex buffer.

INDICES AND INDEX BUFFERS

- Similar to vertices, in order for the GPU to access an array of indices, they need to be placed in a buffer GPU resource

▪ ID3D12Resource

- An index buffer view is represented by the D3D12_INDEX_BUFFER_VIEW structure.

This structure is passed into ID3D12GraphicsCommandList::IASetIndexBuffer.

```
typedef struct D3D12_INDEX_BUFFER_VIEW { D3D12_GPU_VIRTUAL_ADDRESS BufferLocation; UINT SizeInBytes; DXGI_FORMAT Format; } D3D12_INDEX_BUFFER_VIEW;
```

BufferLocation: The GPU virtual address of the index buffer. D3D12_GPU_VIRTUAL_ADDRESS is a typedef'd synonym of UINT64.

SizeInBytes: The size in bytes of the index buffer.

Format: A DXGI_FORMAT-typed value for the index-buffer format.

DXGI_FORMAT_R16_UINT for 16-bit indices or

DXGI_FORMAT_R32_UINT for 32-bit indices

Cube Example

```
std::uint16_t indices[] = {
// front face
0, 1, 2,
0, 2, 3,
// back face
4, 6, 5,
4, 7, 6,
// left face
4, 5, 1,
4, 1, 0,
// right face
3, 2, 6,
3, 6, 7,
// top face
1, 5, 6,
1, 6, 2,
// bottom face
4, 0, 3,
4, 3, 7
};
```

```
const UINT ibByteSize = 36 * sizeof(std::uint16_t);

ComPtr<ID3D12Resource> IndexBufferGPU = nullptr;

ComPtr<ID3D12Resource> IndexBufferUploader = nullptr;

IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),

mCommandList.Get(), indices), ibByteSize,

IndexBufferUploader);

D3D12_INDEX_BUFFER_VIEW ibv;

ibv.BufferLocation = IndexBufferGPU -> GetGPUVirtualAddress();

ibv.Format = DXGI_FORMAT_R16_UINT;

ibv.SizeInBytes = ibByteSize;

mCommandList->IASetIndexBuffer(&ibv);
```

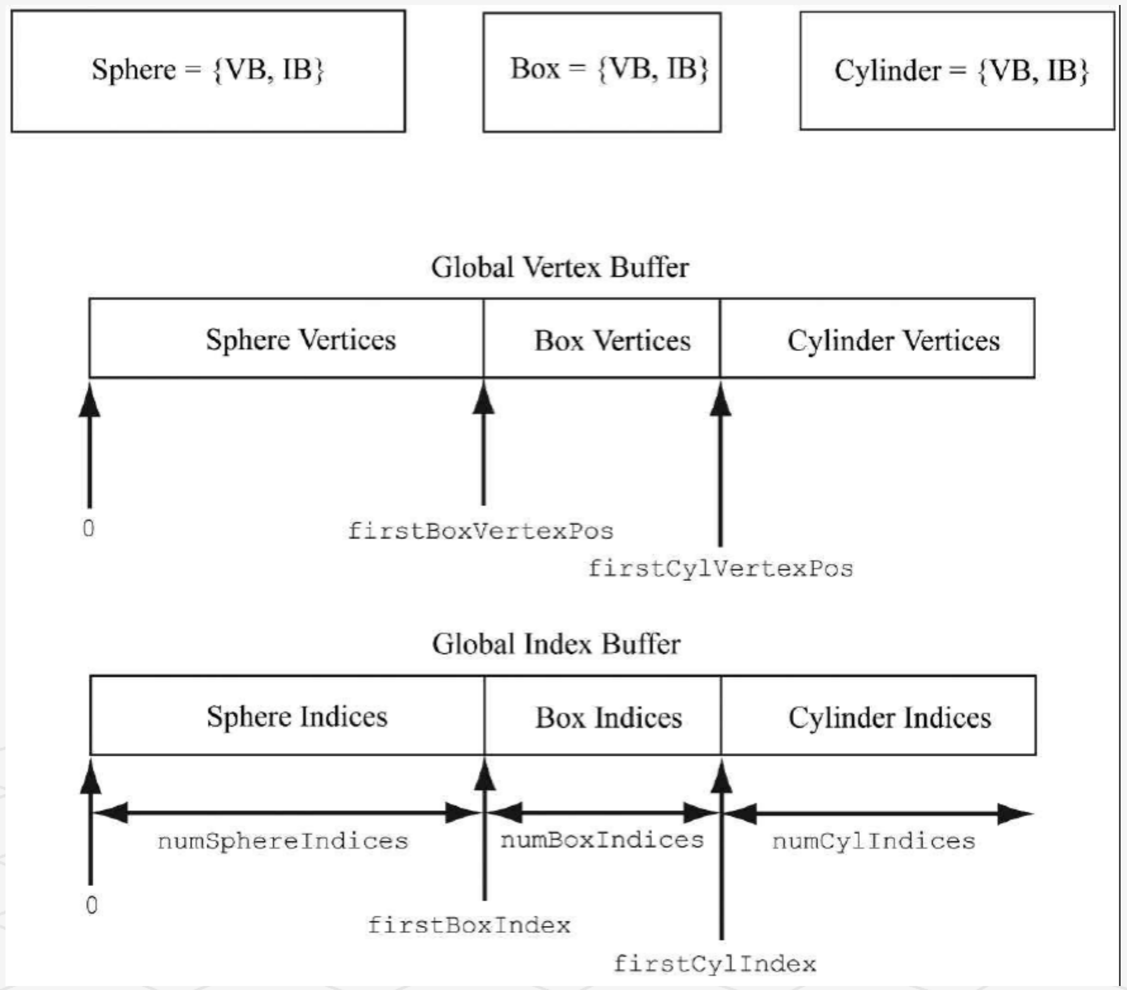
DrawIndexedInstanced

```
void DrawIndexedInstanced(
UINT IndexCountPerInstance,
UINT InstanceCount,
UINT StartIndexLocation,
INT BaseVertexLocation,
UINT StartInstanceLocation)
```

1. IndexCountPerInstance: The number of indices to draw (per instance).
2. InstanceCount: Used for an advanced technique called instancing; for now, set this to 1 as we only draw one instance.
3. StartIndexLocation: Index to an element in the index buffer that marks the starting point from which to begin reading indices.
4. BaseVertexLocation: An integer value to be added to the indices used in this draw call before the vertices are fetched.
5. StartInstanceLocation: Used for an advanced technique called instancing; for now, set this to 0.

concatenate vertex and index buffers

Suppose we have three objects: a sphere, box, and cylinder. At first, each object has its own vertex buffer and its own index buffer. The indices in each local index buffer are relative to the corresponding local vertex buffer. Now suppose that we concatenate the vertices and indices of the sphere, box, and cylinder into one global vertex and index buffer



concatenate vertex and index buffers

Concatenating several vertex buffers into one large vertex buffer, and concatenating several index buffers into one large index buffer.

0, 1, …, numBoxVertices-1

But after the merger, they run from

firstBoxVertexPos,

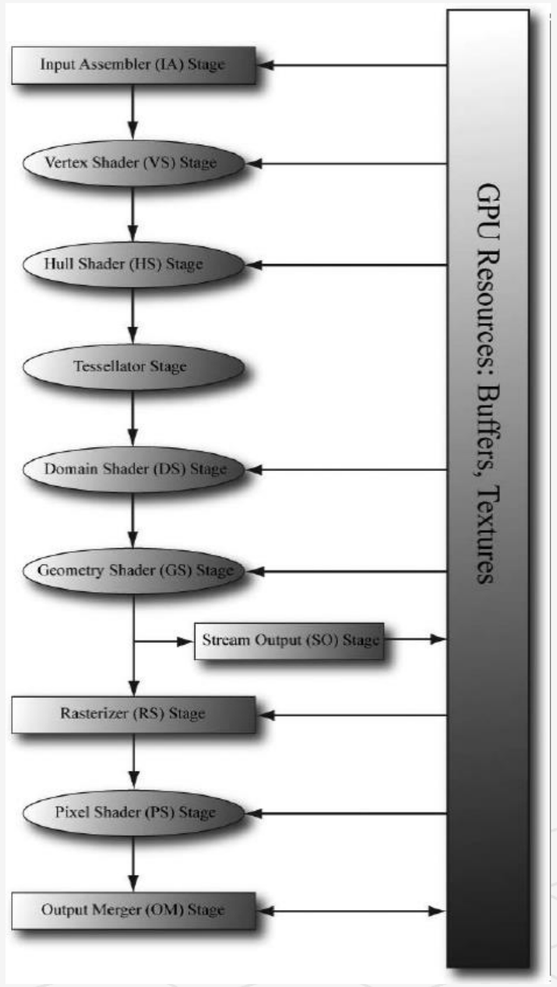
firstBoxVertexPos+1,

…,

firstBoxVertexPos+numBoxVertices-1

```
   mCmdList->DrawIndexedInstanced(
     numSphereIndices, 1, 0, 0, 0);
   mCmdList->DrawIndexedInstanced(
     numBoxIndices, 1, firstBoxIndex, firstBoxVertexPos,
0);
   mCmdList->DrawIndexedInstanced(
     numCylIndices, 1, firstCylIndex, firstCylVertexPos,
0);
```

VERTEX SHADER



```
cbuffer cbPerObject : register(b0)

{

float4x4 gWorldViewProj;

};

void VS(float3 iPosL : POSITION,

float4 iColor : COLOR,

out float4 oPosH : SV_POSITION,

out float4 oColor : COLOR)

{

// Transform to homogeneous clip space.

oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

// Just pass vertex color into the pixel shader.

oColor = iColor;

}
```

Vertex shader

 Shaders are written in a language called the high level shading language (HLSL), which has similar syntax to C++

Appendix B (the textbook) provides a concise reference to the HLSL.

 The vertex shader is the function called VS

 This vertex shader has four parameters

▪ the first two are input parameters

▪ the last two are output parameters (indicated by the out keyword)

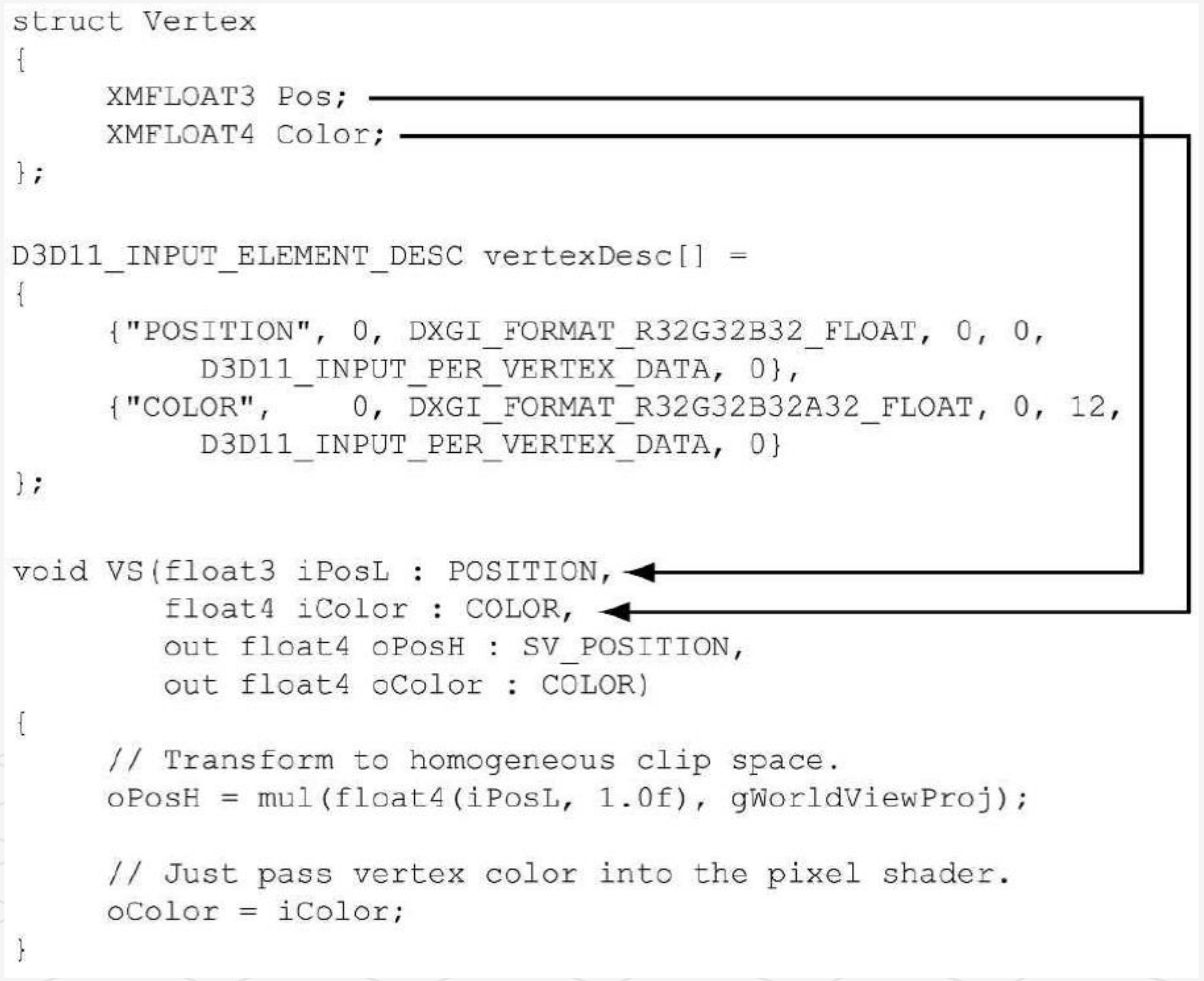
Each vertex element has an associated semantic specified by the

D3D12_INPUT_ELEMENT_DESC array. Each parameter of the vertex shader also

has an attached semantic. The semantics are used to match vertex elements with

vertex shader parameters. The output parameters also have attached semantics (":SV_POSITION" and

":COLOR").



```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};

D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
        D3D11_INPUT_PER_VERTEX_DATA, 0},
    {"COLOR",    0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
        D3D11_INPUT_PER_VERTEX_DATA, 0}
};

void VS(float3 iPosL : POSITION,
        float4 iColor : COLOR,
        out float4 oPosH : SV_POSITION,
        out float4 oColor : COLOR)
{
    // Transform to homogeneous clip space.
    oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

    // Just pass vertex color into the pixel shader.
    oColor = iColor;
}
```

The vertex data and input signature

The vertex data and input signature do not need to match exactly. What is needed is

for the vertex data to provide all the data the vertex shader expects. Therefore, it is

allowed for the vertex data to provide additional data the vertex shader does not use. That is, the following are compatible:

```
//-----
// C++ app code
//-----
struct Vertex
{
XMFLOAT3 Pos;
XMFLOAT4 Color;
XMFLOAT3 Normal;
};
D3D12_INPUT_ELEMENT_DESC desc[] =
{
{“POSITION”, 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_PER_VERTEX_DATA, 0},
{“COLOR”, 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
D3D12_INPUT_PER_VERTEX_DATA, 0},
{ “NORMAL”, 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 28,
D3D12_INPUT_PER_VERTEX_DATA, 0 }
};
//-----
// Vertex shader
//-----
struct VertexIn
{
float3 PosL : POSITION;
float4 Color : COLOR;
};
struct VertexOut
{
float4 PosH : SV_POSITION;
float4 Color : COLOR;
};
VertexOut VS(VertexIn vin) { … }
```

The vertex data and input signature

Now consider the case where the vertex structure and input signature have matching vertex elements, but the types are different for the color attribute.

This is actually legal because Direct3D allows the bits in the input registers to be reinterpreted. However, the VC++ debug output window gives the following warning:

D3D12 WARNING: ID3D11Device::CreateInputLayout: The provided input signature expects to read an element with SemanticName/Index: 'COLOR'/0 and component(s) of the type 'int32'.

```
//-----
// C++ app code
//-----
struct Vertex
{
XMFLOAT3 Pos;
XMFLOAT4 Color;
};
D3D12_INPUT_ELEMENT_DESC desc[] =
{
{"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_PER_VERTEX_DATA, 0},
{"COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
D3D12_INPUT_PER_VERTEX_DATA, 0}
};
//-----
// Vertex shader
//-----
struct VertexIn
{
float3 PosL : POSITION;
int4 Color : COLOR;
};
struct VertexOut
{
float4 PosH : SV_POSITION;
float4 Color : COLOR;
};
VertexOut VS(VertexIn vin) { … }
```

BoxApp::BuildShadersAndInputLayout

```
void BoxApp::BuildShadersAndInputLayout()

{

    HRESULT hr = S_OK;



mvsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr, "VS", "vs_5_0");

mpsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr, "PS", "ps_5_0");



    mInputLayout =

    {

        { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },

        { "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12, D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }

    };

}
```

Color.hlsl

```
// gWorldViewProj lives in a constant buffer
cbuffer cbPerObject : register(b0)
{
float4x4 gWorldViewProj;
};
// Input signature maps to vertex structutre
struct VertexIn
{
float3 PosL  : POSITION;
    float4 Color : COLOR;
};
// Map vertex shadre output to inputs of the next stage
struct VertexOut
{
float4 PosH  : SV_POSITION; //System-value: Homogenous Clip Space
    float4 Color : COLOR;
};
VertexOut VS(VertexIn vin)
{
VertexOut vout;
// Transform to homogeneous clip space.vertex-matrix multiplication
vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

// Just pass vertex color into the pixel shader.
    vout.Color = vin.Color;
    return vout;
}
```

: register (*[shader_profile]*, *Type#[subcomponent]*)

https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-variable-register

Type	Register Description
b	Constant buffer
t	Texture and texture buffer
c	Buffer offset
s	Sampler
u	Unordered Access View

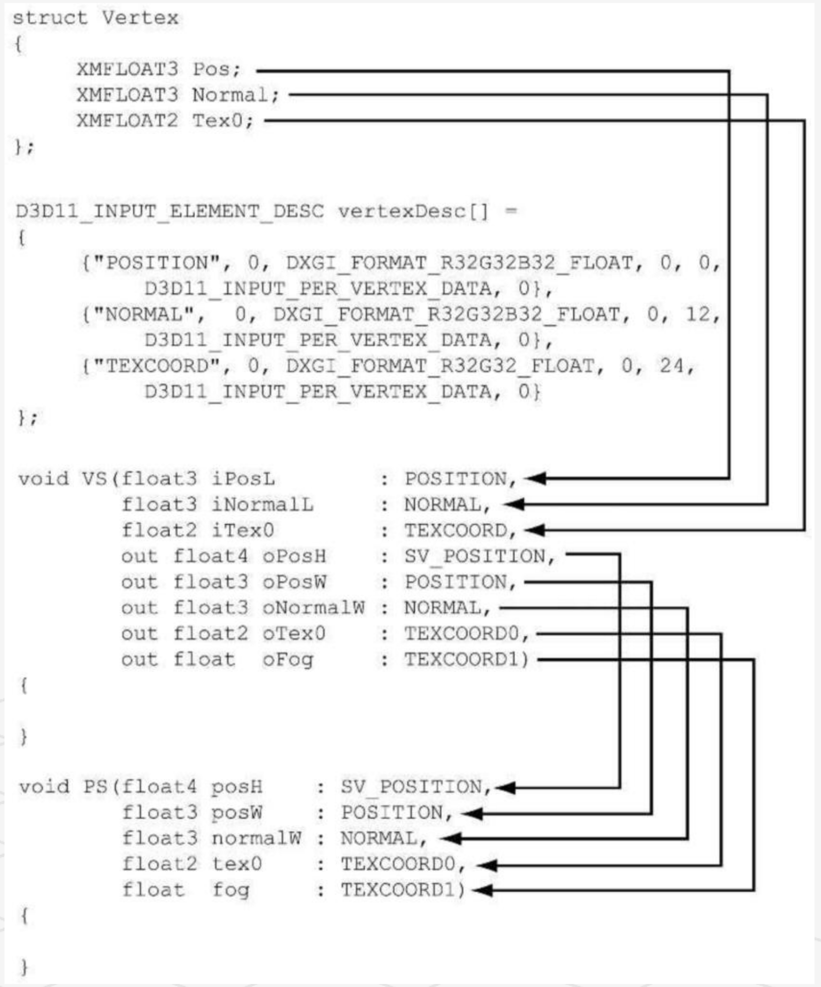
The vertex shader is the function called VS. Note that you can give the vertex shader any valid function name. The HLSL doesn’t have references or pointers, you need to use either structure or out pointers. In HLSL, functions are always inlined.

PIXEL SHADER

During rasterization vertex attributes output from the vertex shader (or geometry shader) are interpolated across the pixels of a triangle. The interpolated values are then fed into the pixel shader as input. Assuming there is no geometry shader.

Each vertex element has an associated semantic specified by the D3D12_INPUT_ELEMENT_DESC array. Each parameter of the vertex shader also has an attached semantic. The semantics are used to match vertex elements with vertex shader parameters. Likewise, each output from the vertex shader has an attached semantic, and each pixel shader input parameter has an attached semantics.

These semantics are used to map vertex shader outputs into the pixel shader input parameters.



```
struct Vertex
{
     XMFLOAT3 Pos;
     XMFLOAT3 Normal;
     XMFLOAT2 Tex0;
};

D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
     {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
          D3D11_INPUT_PER_VERTEX_DATA, 0},
     {"NORMAL",  0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
          D3D11_INPUT_PER_VERTEX_DATA, 0},
     {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
          D3D11_INPUT_PER_VERTEX_DATA, 0}
};

void VS(float3 iPosL        : POSITION,
        float3 iNormalL     : NORMAL,
        float2 iTex0        : TEXCOORD,
        out float4 oPosH    : SV_POSITION,
        out float3 oPosW    : POSITION,
        out float3 oNormalW : NORMAL,
        out float2 oTex0    : TEXCOORD0,
        out float  oFog     : TEXCOORD1)
{

}

void PS(float4 posH     : SV_POSITION,
        float3 posW     : POSITION,
        float3 normalW  : NORMAL,
        float2 tex0     : TEXCOORD0,
        float  fog      : TEXCOORD1)
{

}
```

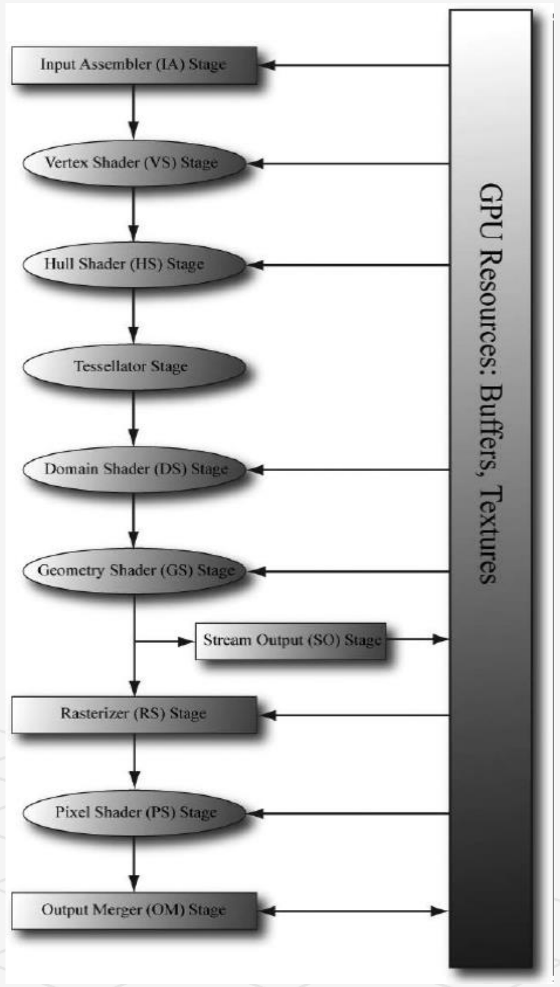
Pixel Shader

- A pixel shader is like a vertex shader in that it is a function executed for each pixel fragment

- Given the pixel shader input, the job of the pixel shader is to calculate a color value for the pixel fragment

- During rasterization vertex attributes output from the vertex shader (or geometry shader) are interpolated across the pixels of a triangle

- The interpolated values are then fed into the pixel shader as input



Color.hlsl

In this example, the pixel shader simply returns the interpolated color value.

Notice that the pixel shader input exactly matches the vertex shader output; this is a requirement.

The pixel shader returns a 4D color value, and the SV_TARGET semantic following the function parameter listing indicates the return value type should match the render target format.

We can equivalently rewrite the pixel shader using input/output structures.

```
float4 PS(float4 posH : SV_POSITION, float4 color :

COLOR) : SV_Target

{

return pin.Color;

}

float4 PS(VertexOut pin) : SV_Target

{

    return pin.Color;

}
```

CONSTANT BUFFERS

- A constant buffer is an example of a GPU resource (ID3D12Resource) whose data contents can be referenced in shader programs

- Unlike vertex and index buffers, constant buffers are usually updated once per frame by the CPU. Therefore, we create constant buffers in an upload heap rather than a default heap so that we can update the contents from the CPU.

- Constant buffers also have the special hardware requirement that their size must be a multiple of the minimum hardware allocation size (256 bytes)

This code refers to a cbuffer object (constant buffer) called cbPerObject. In this example, the constant buffer stores a single 4 × 4 matrix called gWorldViewProj, representing the combined world, view, and projection matrices used to transform a point from local space to homogeneous clip space.

```
cbuffer cbPerObject : register(b0)

{

float4x4 gWorldViewProj;

};
```

In HLSL, a 4 × 4 matrix is declared by the built-in float4x4 type; to declare a 3 × 4 matrix and 2 × 4 matrix, for example, you would use the float3x4 and float2x2 types, respectively.

CONSTANT BUFFERS

Often we will need multiple constant buffers of the same type.

For example, the constant buffer cbPerObject stores constants that vary per object, so if we have *n* objects, then we will need *n* constant buffers of this type. The following code shows how we create a buffer that stores NumElements many constant buffers.

We can think of the mUploadCBuffer as storing an array of constant buffers of type ObjectConstants (with padding to make a multiple of 256 bytes). When it comes time to draw an object, we just bind a constant buffer view (CBV) to a subregion of the buffer that stores the constants for that object. Note that we will often call the buffer mUploadCBuffer a constant buffer since it stores an array of constant buffers.

```
struct ObjectConstants
{
DirectX::XMFLOAT4X4 WorldViewProj =
MathHelper::Identity4x4();
};
UINT elementByteSize =
d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));
ComPtr<ID3D12Resource> mUploadCBuffer;
device->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
D3D12_HEAP_FLAG_NONE,
&CD3DX12_RESOURCE_DESC::Buffer(mElementByteSize* NumElements),
D3D12_RESOURCE_STATE_GENERIC_READ,
nullptr,
IID_PPV_ARGS(&mUploadCBuffer));
```

d3dUtil::CalcConstantBufferByteSize

The utility function d3dUtil::CalcConstantBufferByteSize does the arithmetic to round the byte size of the buffer to be a multiple of the minimum hardware allocation size (256 bytes)

Even though we allocate constant data in multiples of 256, it is not necessary to explicitly pad the corresponding constant data in the HLSL structure because it is done implicitly.

To avoid dealing with rounding constant buffer elements to a multiple of 256 bytes, you could explicitly pad all your constant buffer structures to always be a multiple of 256 bytes.

```
static UINT CalcConstantBufferByteSize(UINT byteSize)
    {
        // Constant buffers must be a multiple of the minimum hardware
        // allocation size (usually 256 bytes).  So round up to nearest
        // multiple of 256.  We do this by adding 255 and then masking off
        // the lower 2 bytes which store all bits < 256.
        // Example: Suppose byteSize = 300.
        // (300 + 255) & ~255
        // 555 & ~255
        // 0x022B & ~0x00ff
        // 0x022B & 0xff00
        // 0x0200
        // 512
        return (byteSize + 255) & ~255;
    }
```

```
// Implicitly padded to 256 bytes.
cbuffer cbPerObject : register(b0)
{
float4x4 gWorldViewProj;
};

// Explicitly padded to 256 bytes.
cbuffer cbPerObject : register(b0)
{
float4x4 gWorldViewProj;
float4x4 Pad0;
float4x4 Pad1;
float4x4 Pad1;
};
```

Shader model 5.1

Direct3D 12 introduced shader model 5.1. Shader model 5.1 has introduced analternative HLSL syntax for defining a constant buffer which looks like this:

```
struct ObjectConstants

{

float4x4 gWorldViewProj;

uint matIndex;

};

ConstantBuffer<ObjectConstants> gObjConstants :

register(b0);
```

Here the data elements of the constant buffer are just defined in a separate structure, and then a constant buffer is created from that structure. Fields of the constant buffer are then accessed in the shader using data member syntax:

```
uint index = gObjConstants.matIndex;
```

Updating Constant Buffers

Because a constant buffer is created with the heap type D3D12_HEAP_TYPE_UPLOAD, we can upload data from the CPU to the constant buffer resource.

```
ComPtr<ID3D12Resource> mUploadBuffer;
BYTE* mMappedData = nullptr;
mUploadBuffer->Map(0, nullptr, reinterpret_cast<void**>(&mMappedData));
memcpy(mMappedData, &data, dataSizeInBytes);
if (mUploadBuffer != nullptr)
mUploadBuffer->Unmap(0, nullptr);
mMappedData = nullptr;
```

It is convenient to build a light wrapper around an upload buffer. We define the following class in *UploadBuffer.h*

```
template<typename T>
class UploadBuffer
{
public:
    UploadBuffer(ID3D12Device* device, UINT elementCount, bool isConstantBuffer) :
        mIsConstantBuffer(isConstantBuffer)
    {
```

BoxApp::OnMouseMove

Typically, the world matrix of an object will change when it moves/rotates/scales, the view matrix changes when the camera moves/rotates, and the projection matrix changes when the window is resized. In our demo, we allow the user to rotate and move the camera with the mouse

```
void BoxApp::OnMouseMove(WPARAM btnState, int x, int y)
{
    if((btnState & MK_LBUTTON) != 0)
    {
        // Make each pixel correspond to a quarter of a degree.
        float dx = XMConvertToRadians(0.25f*static_cast<float>(x - mLastMousePos.x));
        float dy = XMConvertToRadians(0.25f*static_cast<float>(y - mLastMousePos.y));

        // Update angles based on input to orbit camera around box.
        mTheta += dx;
        mPhi += dy;

        // Restrict the angle mPhi.
        mPhi = MathHelper::Clamp(mPhi, 0.1f, MathHelper::Pi - 0.1f);
    }
    else if((btnState & MK_RBUTTON) != 0)
    {
        // Make each pixel correspond to 0.005 unit in the scene.
        float dx = 0.005f*static_cast<float>(x - mLastMousePos.x);
        float dy = 0.005f*static_cast<float>(y - mLastMousePos.y);

        // Update the camera radius based on input.
        mRadius += dx - dy;

        // Restrict the radius.
        mRadius = MathHelper::Clamp(mRadius, 3.0f, 15.0f);
    }

    mLastMousePos.x = x;
    mLastMousePos.y = y;
}
```

BoxApp::Update

we update the combined world-view-projection matrix with the new view matrix every frame in the Update function

```
void BoxApp::Update(const GameTimer& gt)
{
    // Convert Spherical to Cartesian coordinates.
    float x = mRadius*sinf(mPhi)*cosf(mTheta);
    float z = mRadius*sinf(mPhi)*sinf(mTheta);
    float y = mRadius*cosf(mPhi);

    // Build the view matrix.
    XMVECTOR pos = XMVectorSet(x, y, z, 1.0f);
    XMVECTOR target = XMVectorZero();
    XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

    XMMATRIX view = XMMatrixLookAtLH(pos, target, up);
    XMStoreFloat4x4(&mView, view);



    XMMATRIX world = XMLoadFloat4x4(&mWorld);
    XMMATRIX proj = XMLoadFloat4x4(&mProj);
    XMMATRIX worldViewProj = world*view*proj;

// Update the constant buffer with the latest worldViewProj matrix.
ObjectConstants objConstants;
    XMStoreFloat4x4(&objConstants.WorldViewProj, XMMatrixTranspose(worldViewProj));
    mObjectCB->CopyData(0, objConstants);
}
```

Constant Buffer Descriptors

We bind a resource to the rendering pipeline through a descriptor object.

So far we have used descriptors/views for render targets, depth/stencil buffers, and vertex and index buffers.

We also need descriptors to bind constant buffers to the pipeline.

D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV heap can store a mixture of constant buffer, shader resource, and unordered access descriptors.

we specify the D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE flag to indicate that this descriptor will be accessed by shader programs.

In the demo, we have no SRV or UAV descriptors, and we are only going to draw one object; therefore, we only need 1 descriptor in this heap to store 1 CBV.

```
ComPtr<ID3D12DescriptorHeap> mCbvHeap = nullptr;

void BoxApp::BuildDescriptorHeaps()

{

    D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc;

    cbvHeapDesc.NumDescriptors = 1;

    cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;

    cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;

cbvHeapDesc.NodeMask = 0;

    ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&cbvHeapDesc,

        IID_PPV_ARGS(&mCbvHeap)));

}
```

BoxApp::BuildConstantBuffers

A constant buffer view is created by filling out a D3D12_CONSTANT_BUFFER_VIEW_DESC instance and calling ID3D12Device::CreateConstantBufferView in order to create a constant-buffer view for accessing resource data.

```
void CreateConstantBufferView( const
D3D12_CONSTANT_BUFFER_VIEW_DESC *pDesc,
D3D12_CPU_DESCRIPTOR_HANDLE DestDescriptor)
```

The D3D12_CONSTANT_BUFFER_VIEW_DESC structure describes a subset of the constant buffer resource to bind to the HLSL constant buffer structure.

A constant buffer stores an array of per-object constants for *n* objects, but we can get a view to the *i*th object constant data by using the BufferLocation and SizeInBytes.

```
void BoxApp::BuildConstantBuffers()
{
mObjectCB = std::make_unique<UploadBuffer<ObjectConstants>>(md3dDevice.Get(), 1,
true);

UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));

D3D12_GPU_VIRTUAL_ADDRESS cbAddress = mObjectCB->Resource() ->GetGPUVirtualAddress();
    // Offset to the ith object constant buffer in the buffer.
    int boxCBufIndex = 0;
cbAddress += boxCBufIndex*objCBByteSize;

D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc;
cbvDesc.BufferLocation = cbAddress;
cbvDesc.SizeInBytes = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));

md3dDevice->CreateConstantBufferView(
&cbvDesc,
mCbvHeap->GetCPUDescriptorHandleForHeapStart());
}
```

Root Signature and Descriptor Tables

Resources are bound to particular register slots, where they can be accessed by shader programs.

In our box demo, vertex and pixel shader expected only a constant buffer to be bound to register b0.

The *root signature* defines what resources the application will bind to the rendering pipeline before a draw call can be executed and where those resources get mapped to shader input registers.

If we think of the shader programs as a function, and the input resources the shaders expect as function parameters, then the root signature can be thought of as defining a function "signature"

```
void BoxApp::BuildRootSignature()
{
// Root parameter can be a table, root descriptor or root constants.
CD3DX12_ROOT_PARAMETER slotRootParameter[1];

// Create a single descriptor table of CBVs.
CD3DX12_DESCRIPTOR_RANGE cbvTable;

cbvTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);
slotRootParameter[0].InitAsDescriptorTable(1, &cbvTable);

// A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(1, slotRootParameter, 0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

// create a root signature with a single slot which points to a descriptor range consisting of a
single constant buffer
ComPtr<ID3DBlob> serializedRootSig = nullptr;
ComPtr<ID3DBlob> errorBlob = nullptr;
HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc, D3D_ROOT_SIGNATURE_VERSION_1,
serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());

if(errorBlob != nullptr)
{
::OutputDebugStringA((char*)errorBlob->GetBufferPointer());
}
ThrowIfFailed(hr);

ThrowIfFailed(md3dDevice->CreateRootSignature(
0,
serializedRootSig->GetBufferPointer(),
serializedRootSig->GetBufferSize(),
IID_PPV_ARGS(&mRootSignature)));
}
```

SetGraphicsRootDescriptorTable

The root signature only defines what resources the application will bind to the rendering pipeline; it does not actually do any resource binding. Once a root signature has been set with a command list, we use the`ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable` to bind a descriptor table to the pipeline:

1. RootParameterIndex: Index of the root parameter we are setting.

2. BaseDescriptor: Handle to a descriptor in the heap that specifies the first descriptor in the table being set. For example, if the root signature specified that this table had five descriptors, then BaseDescriptor and the next four descriptors in the heap are being set to this root table.

The following code sets the root signature and CBV heap to the command list, and sets the descriptor table identifying the resource we want to bind to the pipeline:

```
void SetGraphicsRootDescriptorTable(

UINT RootParameterIndex,

D3D12_GPU_DESCRIPTOR_HANDLE BaseDescriptor)

mCommandList -> SetGraphicsRootSignature(mRootSignature.Get());

ID3D12DescriptorHeap* descriptorHeaps[] = {mCbvHeap.Get() };

mCommandList -> SetDescriptorHeaps(_countof(descriptorHeaps), descriptorHeaps);

// Offset the CBV we want to use for this draw call.

CD3DX12_GPU_DESCRIPTOR_HANDLE cbv(mCbvHeap ->
GetGPUDescriptorHandleForHeapStart());

cbv.Offset(cbvIndex, mCbvSrvUavDescriptorSize);

mCommandList->SetGraphicsRootDescriptorTable(0, cbv);
```

COMPILING SHADERS

In Direct3D, shader programs must first be compiled to a portable bytecode. The graphics driver will then take this bytecode and compile it again into optimal native instructions for the system's GPU.

`pFileName:` The name of the .hlsl file that contains the HLSL source code we want to compile.

pEntrypoint: The function name of the shader's entry point. A .hlsl can contain multiple shaders programs (e.g., one vertex shader and one pixel shader), so we need to specify the entry point of the particular shader we want to compile.

pTarget: A string specifying the shader program type and version (vs_5_1)

```
HRESULT D3DCompileFromFile(

LPCWSTR pFileName,

const D3D_SHADER_MACRO* pDefines,

ID3DInclude* pInclude,

LPCSTR pEntrypoint,

LPCSTR pTarget,

UINT Flags1,

UINT Flags2,

ID3DBlob** ppCode,

ID3DBlob** ppErrorMsgs);
```

d3dUtil::CompileShader

To support error output, we implement the following helper function to compile shaders at runtime in *d3dUtil.h/.cpp*

HLSL errors and warnings will be returned through the ppErrorMsgs parameter.

For example, if we misspelled the mul function, then we get the following error output to the debug window:

Shaders\color.hlsl(29,14-55): error X3004: undeclared identifier ‘mu’

```
ComPtr<ID3DBlob> d3dUtil::CompileShader(
const std::wstring& filename,
const D3D_SHADER_MACRO* defines,
const std::string& entrypoint,
const std::string& target)
void BoxApp::BuildShadersAndInputLayout()
{
    HRESULT hr = S_OK;


mvsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr, "VS", "vs_5_0");
mpsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr, "PS", "ps_5_0");
```

Offline Compilation

Instead of compiling shaders at runtime, we can compile them offline in a separate step.

1. For complicated shaders, compilation can take a long time. Therefore, compiling offline will make your loading times faster.

2. It is convenient to see shader compilation errors earlier in the build process rather than at runtime.

3. Windows Store apps must use offline compilation.

It is the common practice to use the .cso (compiled shader object) extension for compiled shaders.

To compile shaders offline we use the *FXC* tool that comes with DirectX. To compile a vertex and pixel shader stored in *color.hlsl* with entry points VS and PS, respectively, with debugging we would write:

```
fxc "color.hlsl" /Od /Zi /T vs_5_0 /E "VS" /Fo "color_vs.cso" /Fc
"color_vs.asm"
```

```
fxc "color.hlsl" /Od /Zi /T ps_5_0 /E "PS" /Fo "color_ps.cso" /Fc
"color_ps.asm"
```

To compile a vertex and pixel shader stored in *color.hlsl* with entry points VS and PS, respectively, for *release* we would write:

```
fxc "color.hlsl" /T vs_5_0 /E "VS" /Fo "color_vs.cso" /Fc "color_vs.asm"
```

```
fxc "color.hlsl" /T ps_5_0 /E "PS" /Fo "color_ps.cso" /Fc "color_ps.asm"
```

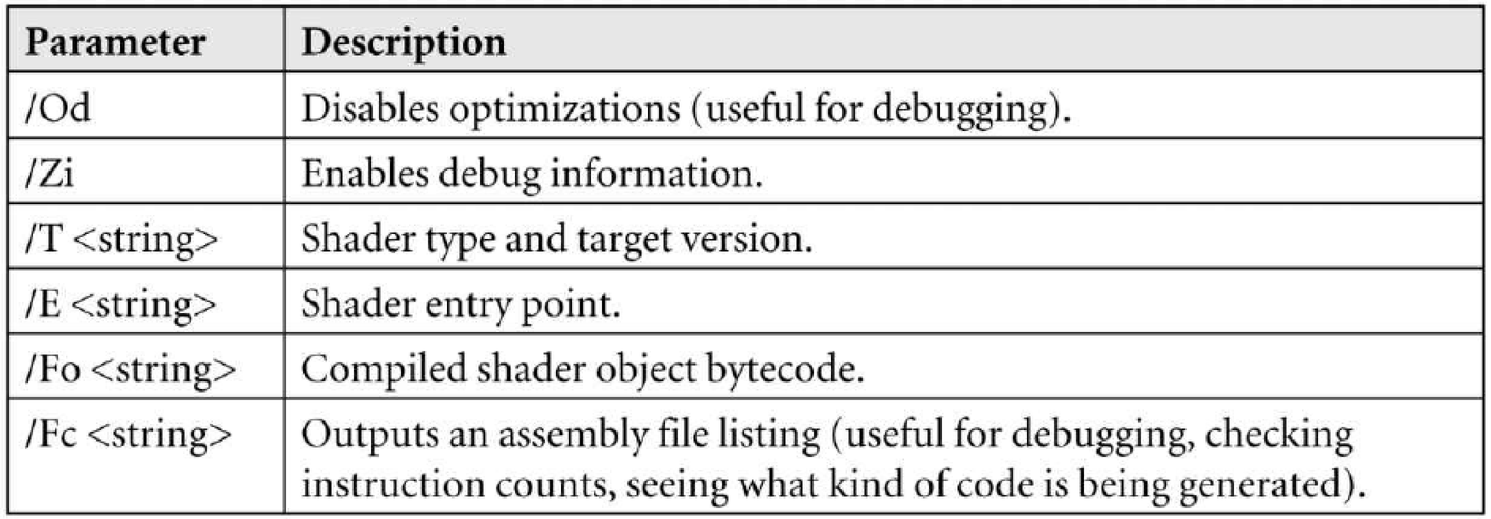
Offline Compiling

Getting the error messages at compile time is much more convenient than runtime.

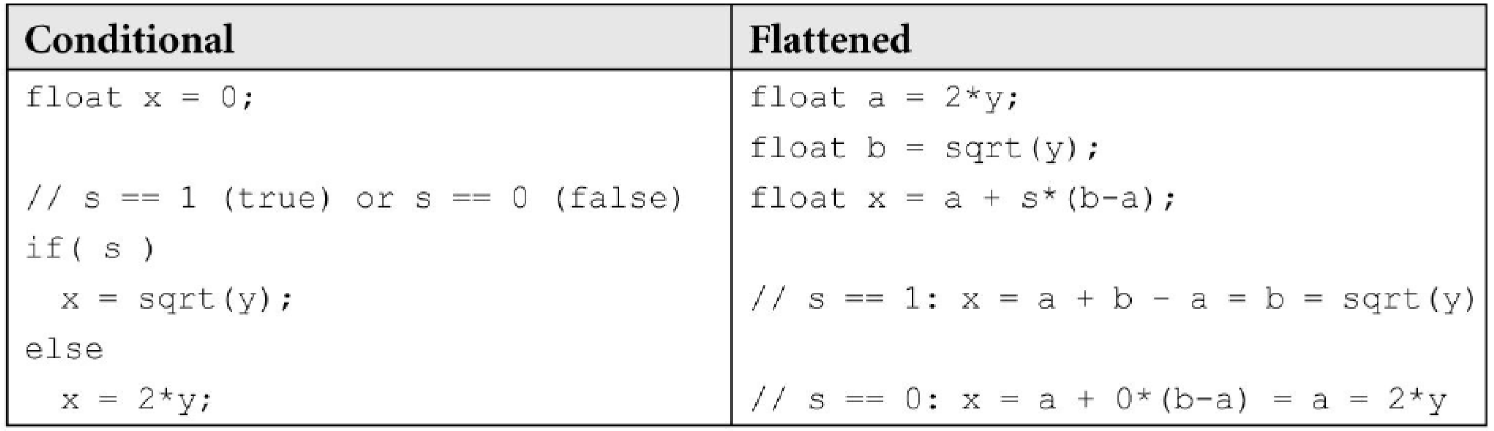
We have shown how to compile our vertex and pixel shaders offline to .cso files. Therefore, we no longer need to do it at runtime (i.e., we do not need to call D3DCompileFromFile).

The /Fc optional parameter to FXC generates the generated portable assembly code. For example, if you have a conditional statement in your HLSL code, then you might expect there to be a branching instruction in the assembly code.

However, we still need to load the compiled shader object bytecode from the .cso files into our app. This can be done using standard C++ file input mechanisms like so: ComPtr<ID3DBlob> d3dUtil::LoadBinary



Parameter	Description
/Od	Disables optimizations (useful for debugging).
/Zi	Enables debug information.
/T <string>	Shader type and target version.
/E <string>	Shader entry point.
/Fo <string>	Compiled shader object bytecode.
/Fc <string>	Outputs an assembly file listing (useful for debugging, checking instruction counts, seeing what kind of code is being generated).



Conditional	Flattened
`float x = 0;`	`float a = 2*y;`
	`float b = sqrt(y);`
`// s == 1 (true) or s == 0 (false)`	`float x = a + s*(b-a);`
`if( s )`	
`  x = sqrt(y);`	`// s == 1: x = a + b - a = b = sqrt(y)`
`else`	
`  x = 2*y;`	`// s == 0: x = a + 0*(b-a) = a = 2*y`

d3dUtil::LoadBinary

```
ComPtr<ID3DBlob> mvsByteCode =

d3dUtil::LoadBinary(L"Shaders\color_vs.cs
o");

ComPtr<ID3DBlob> mpsByteCode =

d3dUtil::LoadBinary(L"Shaders\color_ps.cs
o");
```

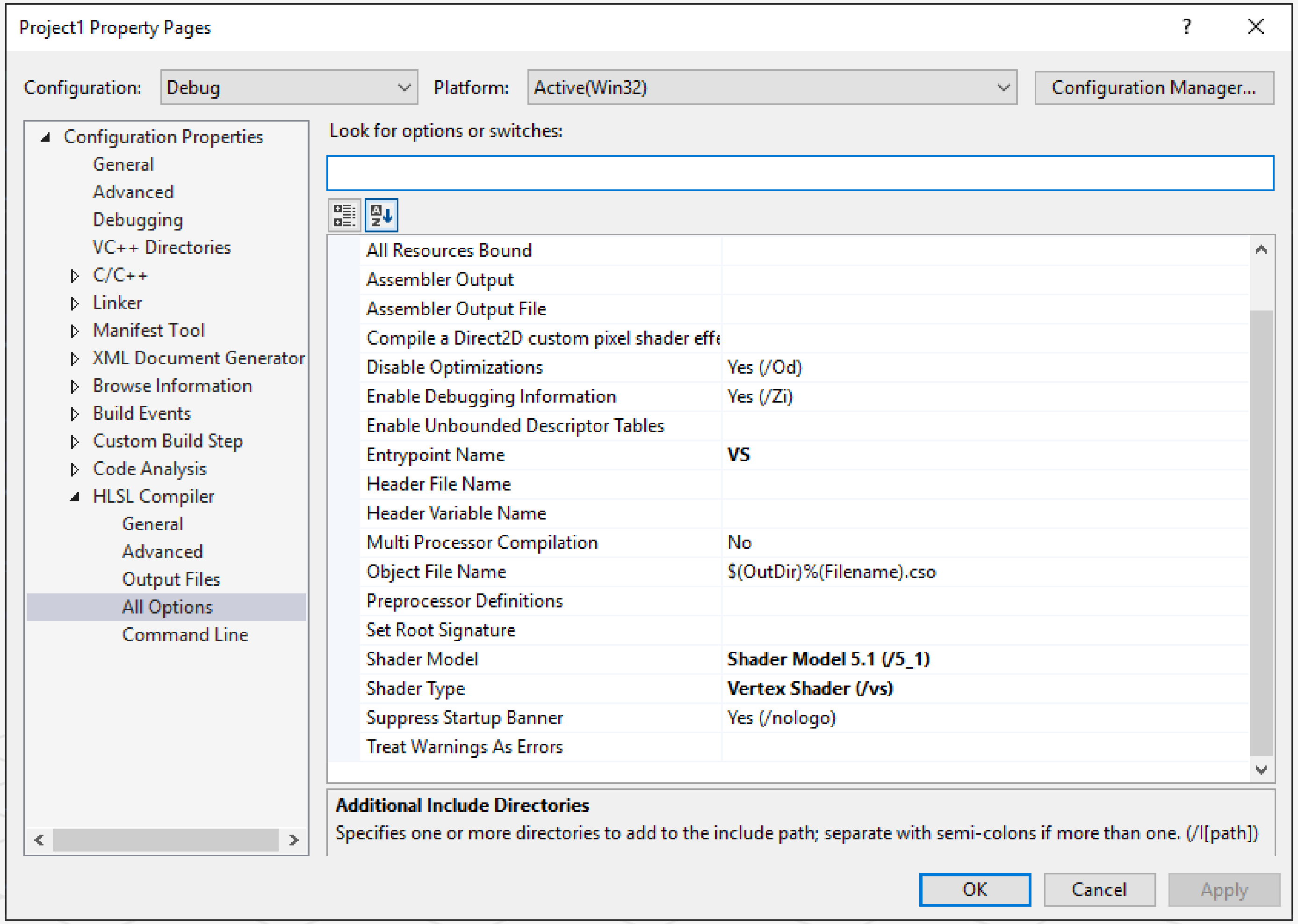
```
ComPtr<ID3DBlob> d3dUtil::LoadBinary(const std::wstring& filename)
{
    std::ifstream fin(filename, std::ios::binary);

    fin.seekg(0, std::ios_base::end);
    std::ifstream::pos_type size = (int)fin.tellg();
    fin.seekg(0, std::ios_base::beg);

    ComPtr<ID3DBlob> blob;
    ThrowIfFailed(D3DCreateBlob(size, blob.GetAddressOf()));

    fin.read((char*)blob->GetBufferPointer(), size);
    fin.close();

    return blob;
}
```

Using Visual Studio to Compile Shaders Offline

You can add .hlsl files to your project, and Visual Studio (VS) will recognize them and provide compilation options .These options provide a UI for the FXC parameters.

When you add a HLSL file to your VS project, it will become part of the build process, and the shader will be compiled with FXC.

One downside to using the VS integrated HLSL support is that it only supports one shader program per file. Therefore, you cannot store both a vertex and pixel shader in one file. Moreover, sometimes we want to compile the same shader program with different preprocessor directives to get different variations of a shader. Again, this will not be possible using the integrated VS support since it is one .cso output per .hlsl input.



Project1 Property Pages

Configuration: Debug   Platform: Active(Win32)   Configuration Manager...

- Configuration Properties
  - General
  - Advanced
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis
  - HLSL Compiler
    - General
    - Advanced
    - Output Files
    - All Options
    - Command Line

Look for options or switches:

All Resources Bound	
Assembler Output	
Assembler Output File	
Compile a Direct2D custom pixel shader effe	
Disable Optimizations	Yes (/Od)
Enable Debugging Information	Yes (/Zi)
Enable Unbounded Descriptor Tables	
Entrypoint Name	**VS**
Header File Name	
Header Variable Name	
Multi Processor Compilation	No
Object File Name	$(OutDir)%(Filename).cso
Preprocessor Definitions	
Set Root Signature	
Shader Model	**Shader Model 5.1 (/5_1)**
Shader Type	**Vertex Shader (/vs)**
Suppress Startup Banner	Yes (/nologo)
Treat Warnings As Errors	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

RASTERIZER STATE

The *rasterizer state* group, represented by the D3D12_RASTERIZER_DESC structure, is used to configure the rasterization stage of the rendering pipeline

1. FillMode: Specify D3D12_FILL_WIREFRAME for wireframe rendering or D3D12_FILL_SOLID for solid rendering. Solid rendering is the default.

2. CullMode: Specify D3D12_CULL_NONE to disable culling,

D3D12_CULL_BACK to cull back-facing triangles, or D3D12_CULL_FRONT to cull front-facing triangles. Back-facing triangles are culled by default.

3. FrontCounterClockwise: Specify false if you want triangles ordered

clockwise (with respect to the camera) to be treated as front-facing and triangles ordered counterclockwise (with respect to the camera) to be treated as back-facing.

Specify true if you want triangles ordered counterclockwise (with respect to the camera) to be treated as front-facing and triangles ordered clockwise (with respect to

the camera) to be treated as back-facing. This state is false by default.

4. ScissorEnable: Specify true to enable the scissor test (§4.3.10) and false to disable it. The default is false.

```
typedef struct D3D12_RASTERIZER_DESC {
D3D12_FILL_MODE FillMode; // Default: D3D12_FILL_SOLID
D3D12_CULL_MODE CullMode; // Default: D3D12_CULL_BACK
BOOL FrontCounterClockwise; // Default: false
INT DepthBias; // Default: 0
FLOAT DepthBiasClamp; // Default: 0.0f
FLOAT SlopeScaledDepthBias; // Default: 0.0f
BOOL DepthClipEnable; // Default: true
BOOL ScissorEnable; // Default: false
BOOL MultisampleEnable; // Default: false
BOOL AntialiasedLineEnable; // Default: false
UINT ForcedSampleCount; // Default: 0
// Default: D3D12_CONSERVATIVE_RASTERIZATION_MODE_OFF
D3D12_CONSERVATIVE_RASTERIZATION_MODE ConservativeRaster;
} D3D12_RASTERIZER_DESC;

Example:
psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
```

```
CD3DX12_RASTERIZER_DESC rsDesc(D3D12_DEFAULT);

rsDesc.FillMode = D3D12_FILL_WIREFRAME;

rsDesc.CullMode = D3D12_CULL_NONE;
```

PIPELINE STATE OBJECT

We have shown, for example, how to describe an input layout description, how to create vertex and pixel shaders, and how to configure the rasterizer state group. However, we have not yet shown how to bind any of these objects to the graphics pipeline for actual use.

Most of the objects that control the state of the graphics pipeline are specified as an aggregate called a *pipeline state object* (PSO), which is represented by the ID3D12PipelineState interface

```
typedef struct D3D12_GRAPHICS_PIPELINE_STATE_DESC
    {
    ID3D12RootSignature *pRootSignature;
    D3D12_SHADER_BYTECODE VS;
    D3D12_SHADER_BYTECODE PS;
    D3D12_SHADER_BYTECODE DS;
    D3D12_SHADER_BYTECODE HS;
    D3D12_SHADER_BYTECODE GS;
    D3D12_STREAM_OUTPUT_DESC StreamOutput;
    D3D12_BLEND_DESC BlendState;
    UINT SampleMask;
    D3D12_RASTERIZER_DESC RasterizerState;
    D3D12_DEPTH_STENCIL_DESC DepthStencilState;
    D3D12_INPUT_LAYOUT_DESC InputLayout;
    D3D12_INDEX_BUFFER_STRIP_CUT_VALUE IBStripCutValue;
    D3D12_PRIMITIVE_TOPOLOGY_TYPE PrimitiveTopologyType;
    UINT NumRenderTargets;
    DXGI_FORMAT RTVFormats[ 8 ];
    DXGI_FORMAT DSVFormat;
    DXGI_SAMPLE_DESC SampleDesc;
    UINT NodeMask;
    D3D12_CACHED_PIPELINE_STATE CachedPSO;
    D3D12_PIPELINE_STATE_FLAGS Flags;
    } D3D12_GRAPHICS_PIPELINE_STATE_DESC;
```

BoxApp::BuildPSO()

By specifying them as an aggregate, Direct3D can validate that all the state is compatible and the driver can generate all the code up front to program the hardware state.

Because PSO validation and creation can be time consuming, PSOs should be generated at initialization time.

Direct3D is basically a state machine. Things stay in their current state until we change them. If some objects you are drawing use one PSO, and other objects you are drawing require a different PSO, then you need to structure your code like this:

// Reset specifies initial PSO.

mCommandList->Reset(mDirectCmdListAlloc.Get(), mPSO1.Get())

/* ...draw objects using PSO 1... */

// Change PSO

mCommandList->SetPipelineState(mPSO2.Get());

/* ...draw objects using PSO 2... */

// Change PSO

mCommandList->SetPipelineState(mPSO3.Get());

/* ...draw objects using PSO 3... */

```
void BoxApp::BuildPSO()
{
    D3D12_GRAPHICS_PIPELINE_STATE_DESC psoDesc;
    ZeroMemory(&psoDesc, sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
    psoDesc.InputLayout = { mInputLayout.data(), (UINT)mInputLayout.size() };
    psoDesc.pRootSignature = mRootSignature.Get();
    psoDesc.VS =
{
reinterpret_cast<BYTE*>(mvsByteCode->GetBufferPointer()),
mvsByteCode->GetBufferSize()
};
    psoDesc.PS =
{
reinterpret_cast<BYTE*>(mpsByteCode->GetBufferPointer()),
mpsByteCode->GetBufferSize()
};
    psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
    psoDesc.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
    psoDesc.DepthStencilState = CD3DX12_DEPTH_STENCIL_DESC(D3D12_DEFAULT);
    psoDesc.SampleMask = UINT_MAX;
    psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
    psoDesc.NumRenderTargets = 1;
    psoDesc.RTVFormats[0] = mBackBufferFormat;
    psoDesc.SampleDesc.Count = m4xMsaaState ? 4 : 1;
    psoDesc.SampleDesc.Quality = m4xMsaaState ? (m4xMsaaQuality - 1) : 0;
    psoDesc.DSVFormat = mDepthStencilFormat;
    ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&psoDesc, IID_PPV_ARGS(&mPSO)));
}
```

GEOMETRY HELPER STRUCTURE

It is helpful to create a structure that groups a vertex and index buffer together to define a group of geometry.

The CPU will need access to the geometry data for things like picking and collision detection.

We use the following MeshGeometry (defined in *d3dUtil.h*) structure throughout the course whenever we define a chunk of geometry.

```
struct MeshGeometry
{
std::string Name;

Microsoft::WRL::ComPtr<ID3DBlob> VertexBufferCPU = nullptr;
Microsoft::WRL::ComPtr<ID3DBlob> IndexBufferCPU  = nullptr;

Microsoft::WRL::ComPtr<ID3D12Resource> VertexBufferGPU = nullptr;
Microsoft::WRL::ComPtr<ID3D12Resource> IndexBufferGPU = nullptr;

Microsoft::WRL::ComPtr<ID3D12Resource> VertexBufferUploader = nullptr;
Microsoft::WRL::ComPtr<ID3D12Resource> IndexBufferUploader = nullptr;

    // Data about the buffers.
UINT VertexByteStride = 0;
UINT VertexBufferByteSize = 0;
DXGI_FORMAT IndexFormat = DXGI_FORMAT_R16_UINT;
UINT IndexBufferByteSize = 0;

std::unordered_map<std::string, SubmeshGeometry> DrawArgs;

D3D12_VERTEX_BUFFER_VIEW VertexBufferView()const
{
D3D12_VERTEX_BUFFER_VIEW vbv;
vbv.BufferLocation = VertexBufferGPU->GetGPUVirtualAddress();
vbv.StrideInBytes = VertexByteStride;
vbv.SizeInBytes = VertexBufferByteSize;

return vbv;
}

D3D12_INDEX_BUFFER_VIEW IndexBufferView()const
{
D3D12_INDEX_BUFFER_VIEW ibv;
ibv.BufferLocation = IndexBufferGPU->GetGPUVirtualAddress();
ibv.Format = IndexFormat;
ibv.SizeInBytes = IndexBufferByteSize;

return ibv;
}
void DisposeUploaders()
{
VertexBufferUploader = nullptr;
IndexBufferUploader = nullptr;
}
};
```

SubmeshGeometry

Defines a subrange of geometry in a MeshGeometry. This is for when multiple geometries are stored in one vertex and index buffer. It provides the offsets and data needed to draw a subset of geometry stores in the vertex and index buffers

```
struct SubmeshGeometry

{

UINT IndexCount = 0;

UINT StartIndexLocation = 0;

INT BaseVertexLocation = 0;

// Bounding box of the geometry defined by this submesh.

DirectX::BoundingBox Bounds;

};
```

DRAWING IN DIRECT3D (PART II)

Objectives:

1. To understand a modification to our rendering process that does not require us to flush the command queue every frame, thereby improving performance.

2. To learn about the two other types of root signature parameter types: root descriptors and root constants.

3. To discover how to procedurally generate and draw common geometric shapes like grids, cylinders, and spheres.

4. To find out how we can animate vertices on the CPU and upload the new vertex positions to the GPU using dynamic vertex buffers.

FRAME Synchronization

The CPU builds and submits command lists and the GPU processes commands in the command queue.

The goal is to keep both CPU and GPU busy to take full advantage of the hardware resources available on the system.

So far in our demos, we have been synchronizing the CPU and GPU once per frame. Why do we need synchronization?

1. ID3D12CommandAllocator represents the allocations of storage for GPU commands. ID3D12CommandAllocator::Reset indicates to re-use the memory that is associated with the command allocator. The command allocator cannot be reset until the GPU is finished executing the commands. If the CPU resets the command allocator in frame $n+1$, but the GPU is still processing commands from frame n, then we would be clearing the commands the GPU is still working on.

2. A constant buffer cannot be updated by the CPU until the GPU has finished executing the drawing commands that reference the constant buffer.

In our box demo, we have been calling D3DApp::FlushCommandQueue at the end of every frame to ensure the GPU has finished executing all the commands for the frame.

Why is this inefficient?

1. At the beginning of a frame, the GPU will not have any commands to process since we waited to empty the command queue. It will have to wait until the CPU builds and submits some commands for execution.

2. At the end of a frame, the CPU is waiting for the GPU to finish processing commands.

FRAME RESOURCES

The frame resources are a circular array of the resources (usually 3) the CPU needs to modify each frame. The CPU will cycle through the frame resource array to get the next available frame resource that is not used by GPU. The CPU will then do any resource updates, and build and submit command lists for frame *n* while the GPU works on previous frames.

Look at Shape Demo → FrameResource.h

```
// Stores the resources needed for the CPU to build the command lists
// for a frame.
struct FrameResource
{
public:

    FrameResource(ID3D12Device* device, UINT passCount, UINT objectCount);
    FrameResource(const FrameResource& rhs) = delete;
    FrameResource& operator=(const FrameResource& rhs) = delete;
    ~FrameResource();

    // We cannot reset the allocator until the GPU is done processing the commands.
    // So each frame needs their own allocator.
    Microsoft::WRL::ComPtr<ID3D12CommandAllocator> CmdListAlloc;

    // We cannot update a cbuffer until the GPU is done processing the commands
    // that reference it.  So each frame needs their own cbuffers.
    std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
    std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;

    // Fence value to mark commands up to this fence point.  This lets us
    // check if these frame resources are still in use by the GPU.
    UINT64 Fence = 0;
};
```

ShapesApp::BuildFrameResources

```
#include "FrameResource.h"


FrameResource::FrameResource(ID3D12Device*
device, UINT passCount, UINT objectCount)
{
    ThrowIfFailed(device->CreateCommandAllocator(
        D3D12_COMMAND_LIST_TYPE_DIRECT,
IID_PPV_ARGS(CmdListAlloc.GetAddressOf())));


    PassCB =
std::make_unique<UploadBuffer<PassConstants>>(dev
ice, passCount, true);
    ObjectCB =
std::make_unique<UploadBuffer<ObjectConstants>>(d
evice, objectCount, true);
}
```

Our application class will then instantiate a vector of three frame resources, and keep member variables to track the current frame resource:

```
const int gNumFrameResources = 3;
std::vector<std::unique_ptr<FrameResource>> mFrameResources;
    FrameResource* mCurrFrameResource = nullptr;
    int mCurrFrameResourceIndex = 0;
void ShapesApp::BuildFrameResources()
{
    for(int i = 0; i < gNumFrameResources; ++i)
    {

mFrameResources.push_back(std::make_unique<FrameResource>(md3dDevice.Get(),
            1, (UINT)mAllRitems.size()));
    }
}
```

RENDER ITEMS

As we begin to draw more objects in our scenes, it is helpful to create a lightweight structure that stores the data needed to draw an object.

We call the set of data needed to submit a full draw call the rendering pipeline a *render item*.

`struct RenderItem` is a Lightweight structure that stores parameters to draw a shape. This will vary from app-to-app.

Our application will maintain lists of render items based on how they need to be drawn; that is, render items that need different PSOs will be kept in different lists.

```
struct RenderItem
{
RenderItem() = default;

    // World matrix of the shape that describes the object's local space
    // relative to the world space, which defines the position, orientation,
    // and scale of the object in the world.
    XMFLOAT4X4 World = MathHelper::Identity4x4();

int NumFramesDirty = gNumFrameResources;

// Index into GPU constant buffer corresponding to the ObjectCB for this render item.
UINT ObjCBIndex = -1;

MeshGeometry* Geo = nullptr;

    // Primitive topology.
    D3D12_PRIMITIVE_TOPOLOGY PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;

    // DrawIndexedInstanced parameters.
    UINT IndexCount = 0;
    UINT StartIndexLocation = 0;
    int BaseVertexLocation = 0;
};

// List of all the render items.
std::vector<std::unique_ptr<RenderItem>> mAllRitems;
// Render items divided by PSO.
std::vector<RenderItem*> mOpaqueRitems;
```

PASS CONSTANTS

We introduced a new constant buffer in our FrameResource class. this buffer stores constant data that is fixed over a given rendering pass such as the eye position, the view and projection matrices, and information about the screen (render target) dimensions; it also includes game timing information, which is useful data to have access to in shader programs

```
 std::unique_ptr<UploadBuffer<PassConstants>> PassCB
= nullptr;

PassCB =
std::make_unique<UploadBuffer<PassConstants>>(device,
passCount, true);
```

```
struct PassConstants
{
    DirectX::XMFLOAT4X4 View = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 InvView = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 Proj = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 InvProj = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 ViewProj = MathHelper::Identity4x4();
    DirectX::XMFLOAT4X4 InvViewProj = MathHelper::Identity4x4();
    DirectX::XMFLOAT3 EyePosW = { 0.0f, 0.0f, 0.0f };
    float cbPerObjectPad1 = 0.0f;
    DirectX::XMFLOAT2 RenderTargetSize = { 0.0f, 0.0f };
    DirectX::XMFLOAT2 InvRenderTargetSize = { 0.0f, 0.0f };
    float NearZ = 0.0f;
    float FarZ = 0.0f;
    float TotalTime = 0.0f;
    float DeltaTime = 0.0f;
};
```

cbuffer cbPass

The per pass constants only need to be updated once per rendering pass, and the object constants only need to change when an object's world matrix changes.

Note that our demos will not necessarily use all this constant data, but it is convenient to have available

We have also modified our per object constant buffer to only store constants that are associated with an object. So far, the only constant data we associate with an object for drawing is its world matrix:

We update our vertex shader accordingly to support these constant buffer changes:

```
cbuffer cbPass : register(b1)
{
    float4x4 gView;
    float4x4 gInvView;
    float4x4 gProj;
    float4x4 gInvProj;
    float4x4 gViewProj;
    float4x4 gInvViewProj;
    float3 gEyePosW;
    float cbPerObjectPad1;
    float2 gRenderTargetSize;
    float2 gInvRenderTargetSize;
    float gNearZ;
    float gFarZ;
    float gTotalTime;
    float gDeltaTime;
};

cbuffer cbPerObject : register(b0)
{
float4x4 gWorld;
};
VertexOut VS(VertexIn vin){
VertexOut vout;
float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
    vout.PosH = mul(posW, gViewProj);
     vout.Color = vin.Color;
    return vout;
}
```

ShapesApp::UpdateMainPassCB & ShapesApp::UpdateObjectCBs

```cpp
void ShapesApp::UpdateMainPassCB(const GameTimer& gt)
{
XMMATRIX view = XMLoadFloat4x4(&mView);
XMMATRIX proj = XMLoadFloat4x4(&mProj);
XMMATRIX viewProj = XMMatrixMultiply(view, proj);
XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);
XMMATRIX invProj = XMMatrixInverse(&XMMatrixDeterminant(proj), proj);
XMMATRIX invViewProj = XMMatrixInverse(&XMMatrixDeterminant(viewProj), viewProj);
XMStoreFloat4x4(&mMainPassCB.View, XMMatrixTranspose(view));
XMStoreFloat4x4(&mMainPassCB.InvView, XMMatrixTranspose(invView));
XMStoreFloat4x4(&mMainPassCB.Proj, XMMatrixTranspose(proj));
XMStoreFloat4x4(&mMainPassCB.InvProj, XMMatrixTranspose(invProj));
XMStoreFloat4x4(&mMainPassCB.ViewProj, XMMatrixTranspose(viewProj));
XMStoreFloat4x4(&mMainPassCB.InvViewProj, XMMatrixTranspose(invViewProj));
mMainPassCB.EyePosW = mEyePos;
mMainPassCB.RenderTargetSize = XMFLOAT2((float)mClientWidth, (float)mClientHeight);
mMainPassCB.InvRenderTargetSize = XMFLOAT2(1.0f / mClientWidth, 1.0f / mClientHeight);
mMainPassCB.NearZ = 1.0f;
mMainPassCB.FarZ = 1000.0f;
mMainPassCB.TotalTime = gt.TotalTime();
mMainPassCB.DeltaTime = gt.DeltaTime();
auto currPassCB = mCurrFrameResource->PassCB.get();
currPassCB->CopyData(0, mMainPassCB);
}
```

```cpp
void ShapesApp::UpdateObjectCBs(const GameTimer& gt)
{
auto currObjectCB = mCurrFrameResource->ObjectCB.get();
for(auto& e : mAllRitems)
{
// Only update the cbuffer data if the constants have changed.
// This needs to be tracked per frame resource.
if(e->NumFramesDirty > 0)
{
XMMATRIX world = XMLoadFloat4x4(&e->World);

ObjectConstants objConstants;
XMStoreFloat4x4(&objConstants.World, XMMatrixTranspose(world));

currObjectCB->CopyData(e->ObjCBIndex, objConstants);

// Next FrameResource need to be updated too.
e->NumFramesDirty--;
}
}
}
```

ShapesApp::BuildRootSignature()

The root signature defines what types of resources are bound to the graphics pipeline. A root signature is configured by the app and links command lists to the resources the shaders require.

The resources that our shaders expect have changed; therefore, we need to update the root signature accordingly to take two descriptor tables (we need two tables because the CBVs will be set at different frequencies—the per pass CBV only needs to be set once per rendering pass while the per object CBV needs to be set per render item):

Descriptor table entries within root signatures contain the descriptor, HLSL shader bind name and visibility flag.

```
void ShapesApp::BuildRootSignature()
{
    CD3DX12_DESCRIPTOR_RANGE cbvTable0;
    cbvTable0.Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 0);

    CD3DX12_DESCRIPTOR_RANGE cbvTable1;
    cbvTable1.Init(D3D12_DESCRIPTOR_RANGE_TYPE_CBV, 1, 1);

// Root parameter can be a table, root descriptor or root constants.
CD3DX12_ROOT_PARAMETER slotRootParameter[2];

// Create root CBVs.
    slotRootParameter[0].InitAsDescriptorTable(1, &cbvTable0);
    slotRootParameter[1].InitAsDescriptorTable(1, &cbvTable1);

// A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(2, slotRootParameter, 0, nullptr,
        D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);

// create a root signature with a single slot which points to a descriptor range consisting of a
single constant buffer
ComPtr<ID3DBlob> serializedRootSig = nullptr;
ComPtr<ID3DBlob> errorBlob = nullptr;
HRESULT hr = D3D12SerializeRootSignature(&rootSigDesc, D3D_ROOT_SIGNATURE_VERSION_1,
serializedRootSig.GetAddressOf(), errorBlob.GetAddressOf());

if(errorBlob != nullptr){::OutputDebugStringA((char*)errorBlob->GetBufferPointer());}
ThrowIfFailed(hr);

ThrowIfFailed(md3dDevice->CreateRootSignature(0,serializedRootSig->GetBufferPointer(),
serializedRootSig->GetBufferSize(),IID_PPV_ARGS(mRootSignature.GetAddressOf())));
}
```

SHAPE GEOMETRY

GeometryGenerator is a utility class for generating simple geometric shapes like grids, sphere, cylinders, and boxes.

```cpp
class GeometryGenerator
{
public:

    using uint16 = std::uint16_t;
    using uint32 = std::uint32_t;

struct Vertex
{
Vertex(){}
        Vertex(
            const DirectX::XMFLOAT3& p,
            const DirectX::XMFLOAT3& n,
            const DirectX::XMFLOAT3& t,
            const DirectX::XMFLOAT2& uv) :
            Position(p),
            Normal(n),
            TangentU(t),
            TexC(uv){}
Vertex(
      float px, float py, float pz,
      float nx, float ny, float nz,
      float tx, float ty, float tz,
      float u, float v) :
      Position(px,py,pz),
      Normal(nx,ny,nz),
      TangentU(tx, ty, tz),
      TexC(u,v){}

        DirectX::XMFLOAT3 Position;
        DirectX::XMFLOAT3 Normal;
        DirectX::XMFLOAT3 TangentU;
        DirectX::XMFLOAT2 TexC;
};
```

Mesh data structure

The MeshData structure is a simple structure nested inside GeometryGenerator that stores a vertex and index list

```
struct MeshData
{
std::vector<Vertex> Vertices;
        std::vector<uint32> Indices32;

        std::vector<uint16>& GetIndices16()
        {
if(mIndices16.empty())
{
mIndices16.resize(Indices32.size());
for(size_t i = 0; i < Indices32.size(); ++i)
mIndices16[i] = static_cast<uint16>(Indices32[i]);
}

return mIndices16;
        }

private:
std::vector<uint16> mIndices16;
};
```

Generating a Cylinder Mesh

We define a cylinder by specifying its bottom and top radii, its height, and the slice and stack count

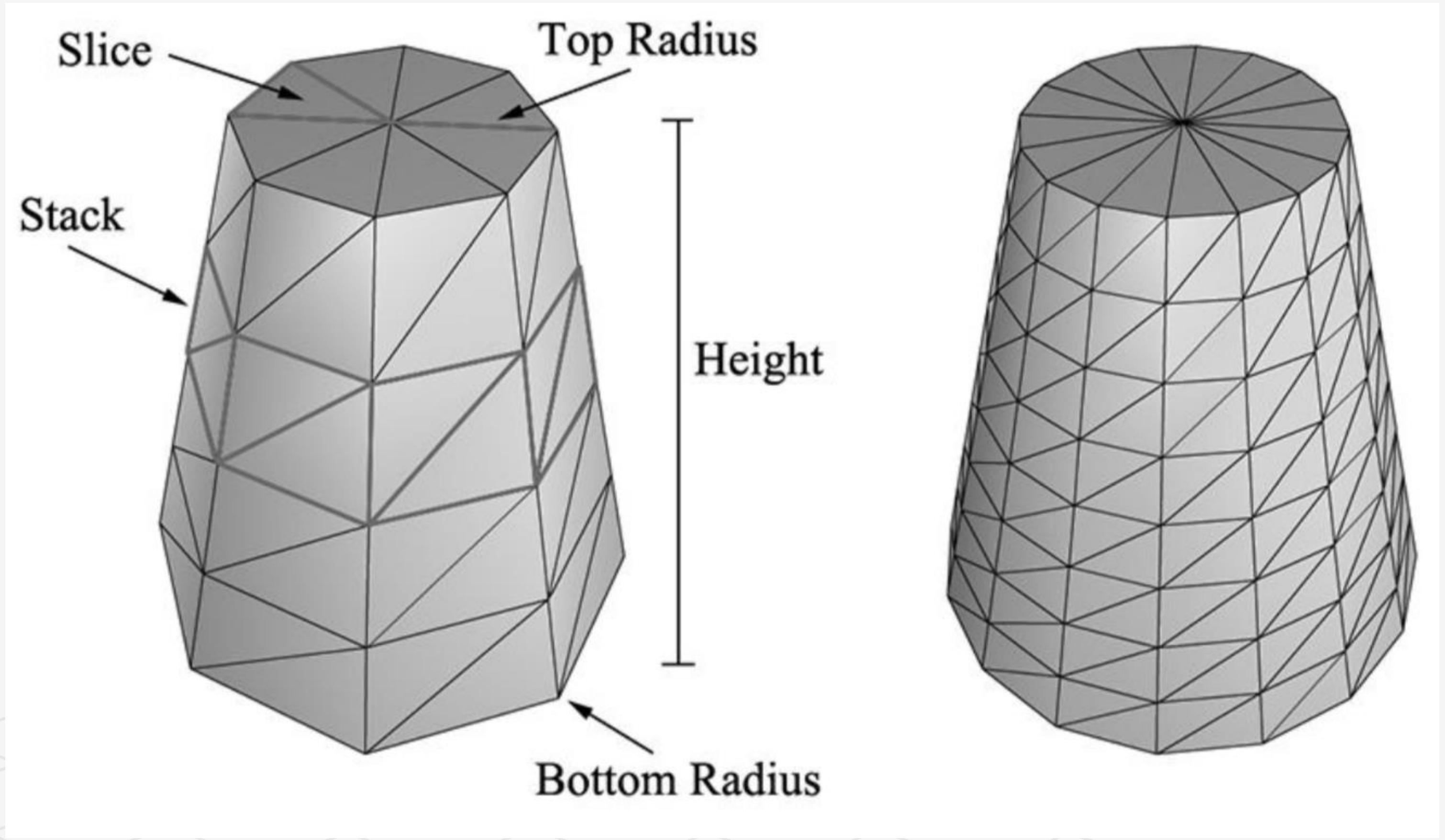
We break the cylinder into three parts:

1) the side geometry

2) the top cap geometry

3) the bottom cap geometry

the cylinder on the left has eight slices and four stacks, and the cylinder on the right has sixteen slices and eight stacks. The slices and stacks control the triangle density. Note that the top and bottom radii can differ so that we can create cone shaped objects, not just "pure" cylinders.



Cylinder Side Geometry

We generate the cylinder centered at the origin, parallel to the *y*-axis.

All the vertices lie on the "rings" of the cylinder, where there are *stackCount* + 1 rings, and each ring has *sliceCount* unique vertices.

The difference in radius between consecutive rings: Δr = *(topRadius – bottomRadius)/stackCount*

If we start at the bottom ring with index 0, then the radius of the *i*th ring is *ri* = *bottomRadius* + *i*Δr and the height of the *i*th ring is: $h_i = -\frac{h}{2} + i\Delta h$

where Δh is the stack height and *h* is the cylinder height.

So the basic idea is to iterate over each ring, and generate the vertices that lie on that ring.

```
GeometryGenerator::MeshData GeometryGenerator::CreateCylinder(float bottomRadius, float topRadius,
float height, uint32 sliceCount, uint32 stackCount)
{
    MeshData meshData;

// Build Stacks.
float stackHeight = height / stackCount;

// Amount to increment radius as we move up each stack level from bottom to top.
float radiusStep = (topRadius - bottomRadius) / stackCount;

uint32 ringCount = stackCount+1;

// Compute vertices for each stack ring starting at the bottom and moving up.
for(uint32 i = 0; i < ringCount; ++i)
{
float y = -0.5f*height + i*stackHeight;
float r = bottomRadius + i*radiusStep;

// vertices of ring
float dTheta = 2.0f*XM_PI/sliceCount;
for(uint32 j = 0; j <= sliceCount; ++j)
{
Vertex vertex;

float c = cosf(j*dTheta);
float s = sinf(j*dTheta);

vertex.Position = XMFLOAT3(r*c, y, r*s);

vertex.TexC.x = (float)j/sliceCount;
vertex.TexC.y = 1.0f - (float)i/stackCount;
```

Cylinder - Side Geometry

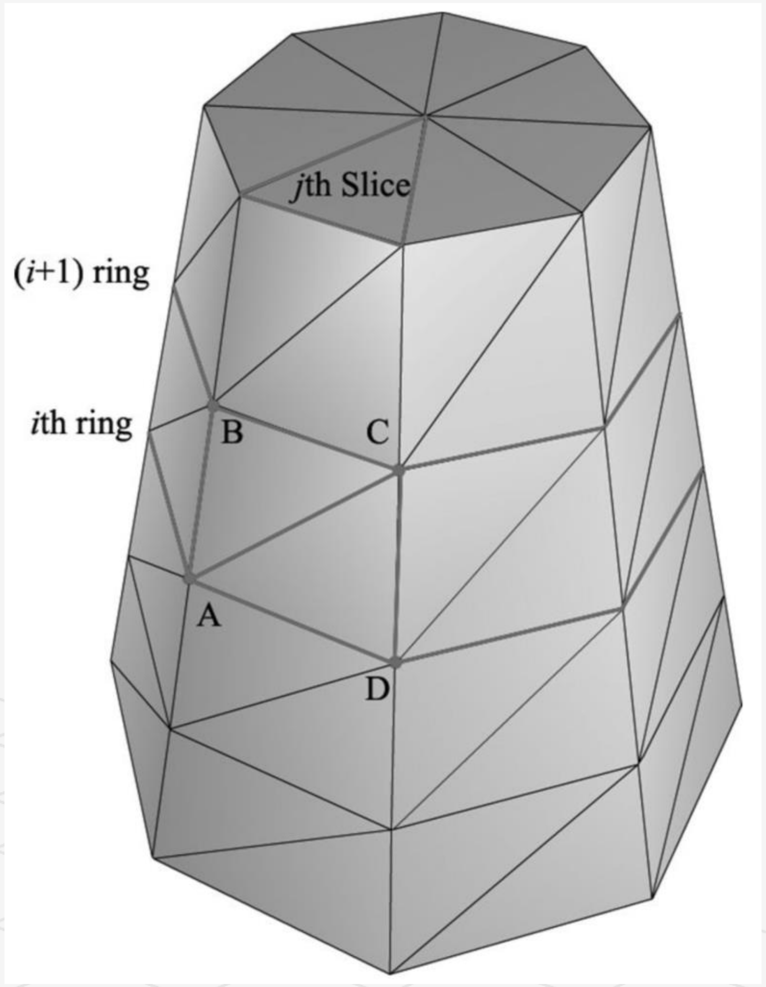
The vertices *A*, *B*, *C*, *D* contained in the *i*th and *i* + 1th ring, and *j*th slice.

There is a quad (two triangles) for each slice in every stack. This figure shows that the indices for the *i*th stack and *j*th slice are given by:

$$\Delta ABC = \left(i \cdot n + j, (i+1) \cdot n + j, (i+1) \cdot n + j + 1\right)$$

$$\Delta ACD = \left(i \cdot n + j, (i+1) \cdot n + j + 1, i \cdot n + j + 1\right)$$

where *n* is the number of vertices per ring. So the key idea is to loop over every slice in every stack, and apply the above formulas.



Cap Geometry

```
void GeometryGenerator::BuildCylinderBottomCap(float bottomRadius, float topRadius, float height,
   uint32 sliceCount, uint32 stackCount, MeshData& meshData){
// Build bottom cap.
uint32 baseIndex = (uint32)meshData.Vertices.size();
float y = -0.5f*height;
// vertices of ring
float dTheta = 2.0f*XM_PI/sliceCount;
for(uint32 i = 0; i <= sliceCount; ++i){
float x = bottomRadius*cosf(i*dTheta);
float z = bottomRadius*sinf(i*dTheta);
// Scale down by the height to try and make top cap texture coord area  proportional to base.
float u = x/height + 0.5f;
float v = z/height + 0.5f;
meshData.Vertices.push_back( Vertex(x, y, z, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, u, v) );}
// Cap center vertex.
meshData.Vertices.push_back( Vertex(0.0f, y, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.5f,0.5f));
// Cache the index of center vertex.
uint32 centerIndex = (uint32)meshData.Vertices.size()-1;
for(uint32 i = 0; i < sliceCount; ++i){
meshData.Indices32.push_back(centerIndex);
meshData.Indices32.push_back(baseIndex + i);
meshData.Indices32.push_back(baseIndex + i+1);}}
```

```
void GeometryGenerator::BuildCylinderTopCap(float bottomRadius, float topRadius, float
height,
uint32 sliceCount, uint32 stackCount, MeshData& meshData)
{
uint32 baseIndex = (uint32)meshData.Vertices.size();

float y = 0.5f*height;
float dTheta = 2.0f*XM_PI/sliceCount;

// Duplicate cap ring vertices because the texture coordinates and normals differ.
for(uint32 i = 0; i <= sliceCount; ++i)
{
float x = topRadius*cosf(i*dTheta);
float z = topRadius*sinf(i*dTheta);

// Scale down by the height to try and make top cap texture coord area
// proportional to base.
float u = x/height + 0.5f;
float v = z/height + 0.5f;

meshData.Vertices.push_back( Vertex(x, y, z, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f, u, v) );
}

// Cap center vertex.
meshData.Vertices.push_back( Vertex(0.0f, y, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 0.0f,
0.5f, 0.5f) );

// Index of center vertex.
uint32 centerIndex = (uint32)meshData.Vertices.size()-1;

for(uint32 i = 0; i < sliceCount; ++i)
{
meshData.Indices32.push_back(centerIndex);
meshData.Indices32.push_back(baseIndex + i+1);
meshData.Indices32.push_back(baseIndex + i);
}
}
```

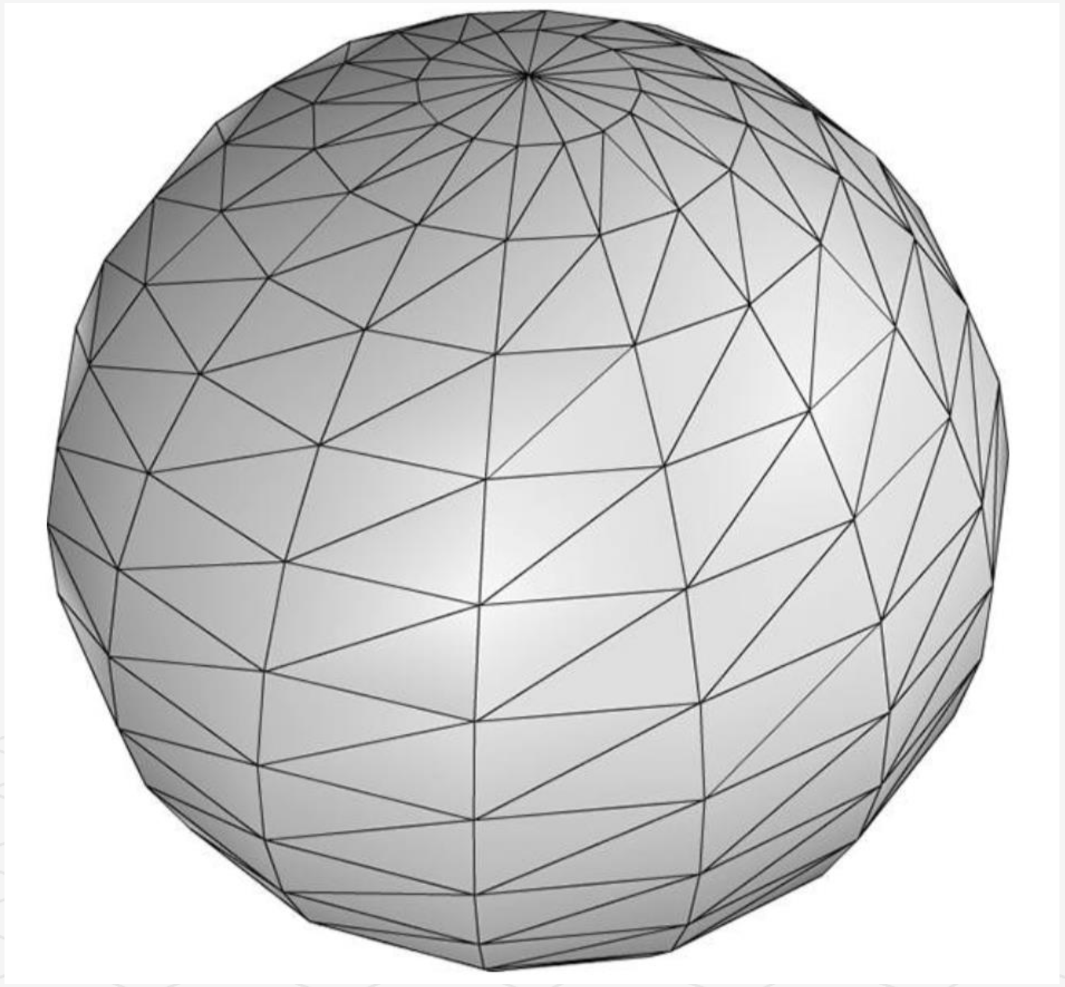
Generating a Sphere Mesh

We define a sphere by specifying its radius, and the slice and stack count

The algorithm for generating the sphere is very similar to that of the cylinder, except that the radius per ring changes is a nonlinear way based on trigonometric functions.

```
GeometryGenerator::MeshData
GeometryGenerator::CreateSphere(float radius, uint32
sliceCount, uint32 stackCount)

{

    MeshData meshData;
```

Note that we can apply a nonuniform scaling world transformation to transform a sphere into an ellipsoid.

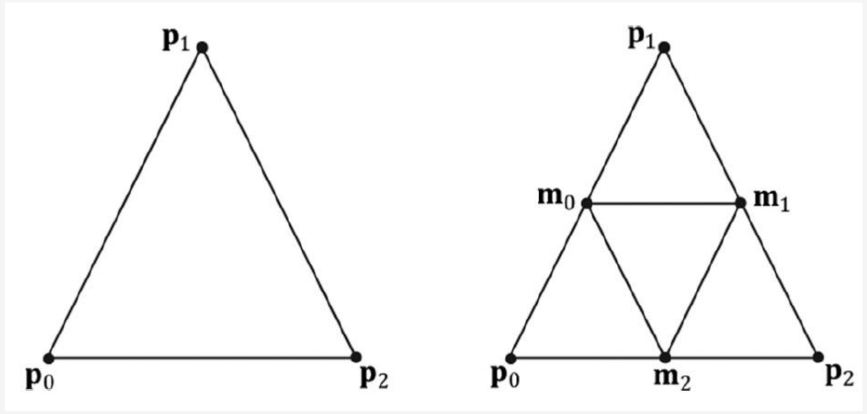


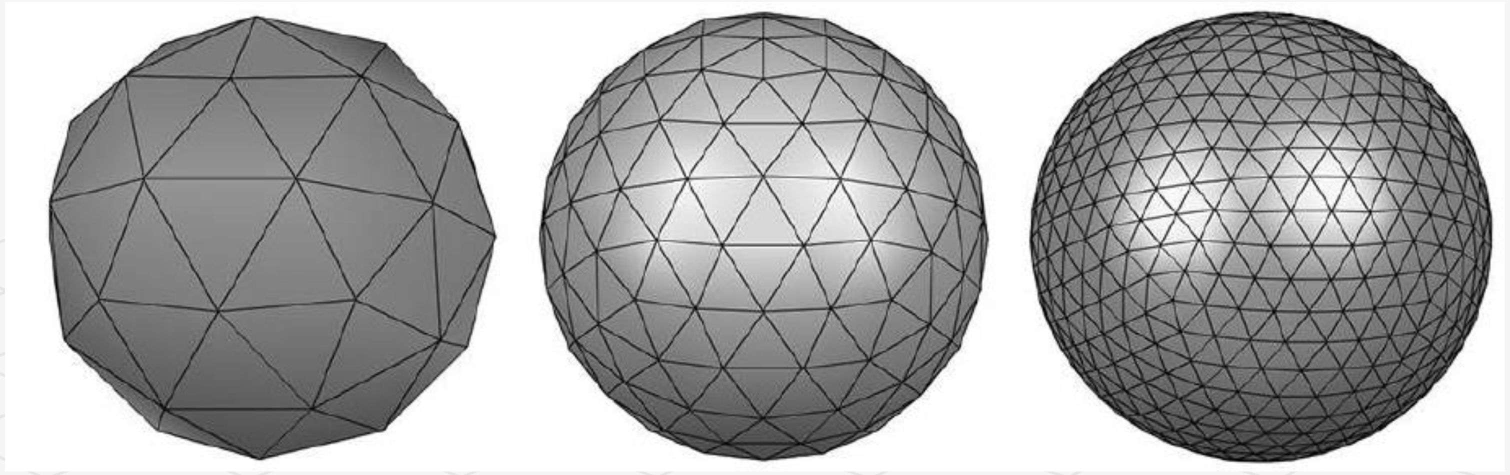
Generating a Geosphere Mesh

The triangles of the sphere do not have equal areas. This can be undesirable for some situations. A geosphere approximates a sphere using triangles with almost equal areas as well as equal side lengths

To generate a geosphere, we start with an icosahedron, subdivide the triangles, and then project the new vertices onto the sphere with the given radius. We can repeat this process to improve the tessellation.

A triangle can be subdivided into four equal sized triangles. The new vertices are found just by taking the midpoints along the edges of the original triangle. The new vertices can then be projected onto a sphere of radius *r* by projecting the vertices onto the unit sphere and then scalar multiplying by: $r : \mathbf{v}' = r\frac{\mathbf{v}}{\|\mathbf{v}\|}$





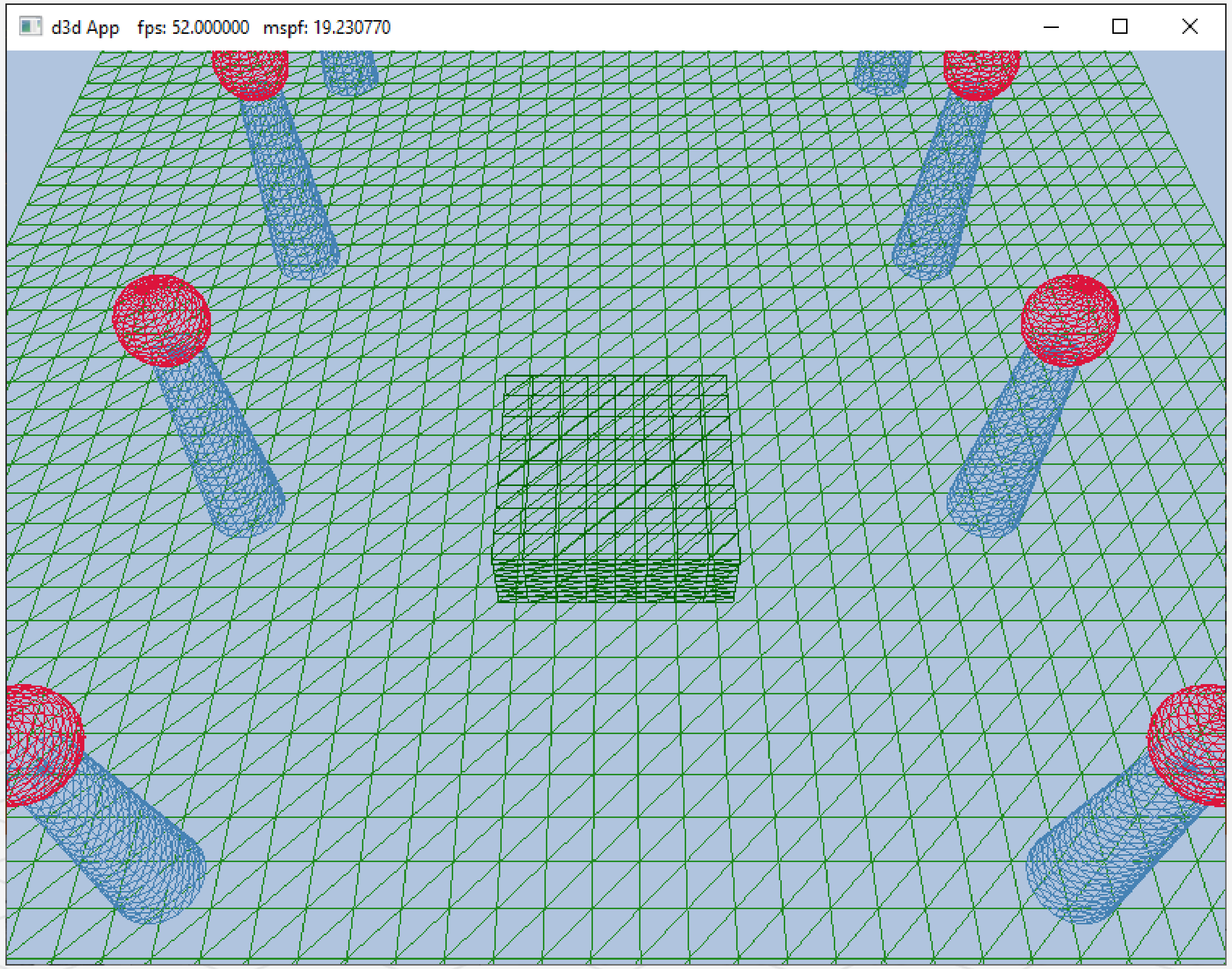
Vertex and Index Buffers

Even though we draw multiple spheres and cylinders in this demo, we only need one copy of the sphere and cylinder geometry.

We simply redraw the same sphere and cylinder mesh multiple times, but with different world matrices; this is an example of *instancing* geometry, which saves memory: ID3D12CommandList::DrawIndexedInstanced

We pack all the mesh vertices and indices into one vertex and index buffer. This is done by concatenating the vertex and index arrays. This means that when we draw an object, we are only drawing a subset of the vertex and index buffers.

We need to know the starting index to the object in the concatenated index buffer, its index count, and we need to know the base vertex location



ShapesApp::BuildShapeGeometry

```cpp
void ShapesApp::BuildShapeGeometry()
{
    GeometryGenerator geoGen;
GeometryGenerator::MeshData box = geoGen.CreateBox(1.5f, 0.5f, 1.5f, 3);
GeometryGenerator::MeshData grid = geoGen.CreateGrid(20.0f, 30.0f, 60, 40);
GeometryGenerator::MeshData sphere = geoGen.CreateSphere(0.5f, 20, 20);
GeometryGenerator::MeshData cylinder = geoGen.CreateCylinder(0.5f, 0.3f, 3.0f, 20,
20);

//
// We are concatenating all the geometry into one big vertex/index buffer.  So
// define the regions in the buffer each submesh covers.
//

// Cache the vertex offsets to each object in the concatenated vertex buffer.
UINT boxVertexOffset = 0;
UINT gridVertexOffset = (UINT)box.Vertices.size();
UINT sphereVertexOffset = gridVertexOffset + (UINT)grid.Vertices.size();
UINT cylinderVertexOffset = sphereVertexOffset + (UINT)sphere.Vertices.size();

// Cache the starting index for each object in the concatenated index buffer.
UINT boxIndexOffset = 0;
UINT gridIndexOffset = (UINT)box.Indices32.size();
UINT sphereIndexOffset = gridIndexOffset + (UINT)grid.Indices32.size();
UINT cylinderIndexOffset = sphereIndexOffset + (UINT)sphere.Indices32.size();
```

```cpp
// Define the SubmeshGeometry that cover different
    // regions of the vertex/index buffers.

SubmeshGeometry boxSubmesh;
boxSubmesh.IndexCount = (UINT)box.Indices32.size();
boxSubmesh.StartIndexLocation = boxIndexOffset;
boxSubmesh.BaseVertexLocation = boxVertexOffset;

SubmeshGeometry gridSubmesh;
gridSubmesh.IndexCount = (UINT)grid.Indices32.size();
gridSubmesh.StartIndexLocation = gridIndexOffset;
gridSubmesh.BaseVertexLocation = gridVertexOffset;

SubmeshGeometry sphereSubmesh;
sphereSubmesh.IndexCount = (UINT)sphere.Indices32.size();
sphereSubmesh.StartIndexLocation = sphereIndexOffset;
sphereSubmesh.BaseVertexLocation = sphereVertexOffset;

SubmeshGeometry cylinderSubmesh;
cylinderSubmesh.IndexCount =
(UINT)cylinder.Indices32.size();
cylinderSubmesh.StartIndexLocation = cylinderIndexOffset;
cylinderSubmesh.BaseVertexLocation = cylinderVertexOffset;
```

std::unordered_map

Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity (e.g. O(1)).

"Constant time" means that the operation will execute in an amount of time (or memory space - that's another thing often measured) **independent** of the input size

It is cumbersome to create a new variable name for each geometry, PSO, texture, shader, etc., so we use unordered maps for constant time lookup and reference our objects by name. Here are some more examples:

```cpp
#include <iostream>
#include <string>
#include <unordered_map>

int main()
{
// Create an unordered_map of three strings (that map to strings)
std::unordered_map<std::string, std::string> u = {
{"RED","#FF0000"},
{"GREEN","#00FF00"},
{"BLUE","#0000FF"}
};

// Iterate and print keys and values of unordered_map
for (const auto& n : u) {
std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
}

// Add two new entries to the unordered_map
u["BLACK"] = "#000000";
u["WHITE"] = "#FFFFFF";

// Output values by key
std::cout << "The HEX of color RED is:[" << u["RED"] << "]\n";
std::cout << "The HEX of color BLACK is:[" << u["BLACK"] << "]\n";

return 0;
}
```

```cpp
std::unordered_map<std::string, std::unique_ptr<MeshGeometry>> mGeometries;
std::unordered_map<std::string, ComPtr<ID3DBlob>> mShaders;
std::unordered_map<std::string, ComPtr<ID3D12PipelineState>> mPSOs;
```

Render Items

We now define our scene render items.

Observe how all the render items share the same MeshGeometry, and we use the DrawArgs to get the DrawIndexedInstanced parameters to draw a subregion of the vertex/index buffers.

```
// List of all the render items.
std::vector<std::unique_ptr<RenderItem>> mAllRitems;

// Render items divided by PSO.
std::vector<RenderItem*> mOpaqueRitems;

void ShapesApp::BuildRenderItems()
{
auto boxRitem = std::make_unique<RenderItem>();
XMStoreFloat4x4(&boxRitem->World, XMMatrixScaling(2.0f, 2.0f,
2.0f)*XMMatrixTranslation(0.0f, 0.5f, 0.0f));
boxRitem->ObjCBIndex = 0;
boxRitem->Geo = mGeometries["shapeGeo"].get();
boxRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
boxRitem->IndexCount = boxRitem->Geo->DrawArgs["box"].IndexCount;
boxRitem->StartIndexLocation = boxRitem->Geo->DrawArgs["box"].StartIndexLocation;
boxRitem->BaseVertexLocation = boxRitem->Geo->DrawArgs["box"].BaseVertexLocation;
mAllRitems.push_back(std::move(boxRitem));

……
// All the render items are opaque.
for(auto& e : mAllRitems)
mOpaqueRitems.push_back(e.get());
```

Frame Resources

we have a vector of FrameResources, and each FrameResource has an upload buffer for storing the pass constants and constant buffers for every render item in the scene.

If we have 3 frame resources and *n* render items, then we have three 3*n* object constant buffers and 3 pass constant buffers. Hence we need 3(*n*+1) constant buffer views (CBVs).

Thus we will need to modify our CBV heap to include the additional descriptors:

```
// We cannot update a cbuffer until the GPU is done processing the commands
// that reference it.  So each frame needs their own cbuffers.
    std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
    std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;

void ShapesApp::BuildDescriptorHeaps()
{
    UINT objCount = (UINT)mOpaqueRitems.size();

    // Need a CBV descriptor for each object for each frame resource,
    // +1 for the perPass CBV for each frame resource.
    UINT numDescriptors = (objCount+1) * gNumFrameResources;

    // Save an offset to the start of the pass CBVs.  These are the last descriptors.
    mPassCbvOffset = objCount * gNumFrameResources;

    D3D12_DESCRIPTOR_HEAP_DESC cbvHeapDesc;
    cbvHeapDesc.NumDescriptors = numDescriptors;
    cbvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
    cbvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
    cbvHeapDesc.NodeMask = 0;
    ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&cbvHeapDesc,
        IID_PPV_ARGS(&mCbvHeap)));
}
```

Constant Buffer Views

Now, we can populate the CBV heap with the following code where descriptors 0 to *n*-1 contain the object CBVs for the 0th frame resource, descriptors *n* to 2*n*−1 contains the object CBVs for 1st frame resource, descriptors 2*n* to 3*n*−1 contain the objects CBVs for the 2nd frame resource, and descriptors 3*n*, 3*n*+1, and 3*n*+2 contain the pass CBVs for the 0th, 1st, and 2nd frame resource, respectively:

```
void ShapesApp::BuildConstantBufferViews()
{
    UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));

    UINT objCount = (UINT)mOpaqueRitems.size();

    // Need a CBV descriptor for each object for each frame resource.
    for(int frameIndex = 0; frameIndex < gNumFrameResources; ++frameIndex)
    {
        auto objectCB = mFrameResources[frameIndex]->ObjectCB->Resource();
        for(UINT i = 0; i < objCount; ++i)
        {
            D3D12_GPU_VIRTUAL_ADDRESS cbAddress = objectCB->GetGPUVirtualAddress();

            // Offset to the ith object constant buffer in the buffer.
            cbAddress += i*objCBByteSize;

            // Offset to the object cbv in the descriptor heap.
            int heapIndex = frameIndex*objCount + i;
    auto handle = CD3DX12_CPU_DESCRIPTOR_HANDLE(mCbvHeap>GetCPUDescriptorHandleForHeapStart());
            handle.Offset(heapIndex, mCbvSrvUavDescriptorSize);

            D3D12_CONSTANT_BUFFER_VIEW_DESC cbvDesc;
            cbvDesc.BufferLocation = cbAddress;
            cbvDesc.SizeInBytes = objCBByteSize;

            md3dDevice->CreateConstantBufferView(&cbvDesc, handle);
```

Drawing the Scene

The only tricky part is offsetting to the correct CBV in the heap for the object we want to draw. Notice how a render item stores an index to the constant buffer that is associated with the render item.

```
void ShapesApp::DrawRenderItems(ID3D12GraphicsCommandList* cmdList, const
std::vector<RenderItem*>& ritems)
{
    UINT objCBByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));

auto objectCB = mCurrFrameResource->ObjectCB->Resource();

    // For each render item...
    for(size_t i = 0; i < ritems.size(); ++i)
    {
        auto ri = ritems[i];

        cmdList->IASetVertexBuffers(0, 1, &ri->Geo->VertexBufferView());
        cmdList->IASetIndexBuffer(&ri->Geo->IndexBufferView());
        cmdList->IASetPrimitiveTopology(ri->PrimitiveType);

        // Offset to the CBV in the descriptor heap for this object and for this frame
resource.
        UINT cbvIndex = mCurrFrameResourceIndex*(UINT)mOpaqueRitems.size() + ri-
>ObjCBIndex;
        auto cbvHandle = CD3DX12_GPU_DESCRIPTOR_HANDLE(mCbvHeap-
>GetGPUDescriptorHandleForHeapStart());
        cbvHandle.Offset(cbvIndex, mCbvSrvUavDescriptorSize);

        cmdList->SetGraphicsRootDescriptorTable(0, cbvHandle);

        cmdList->DrawIndexedInstanced(ri->IndexCount, 1, ri->StartIndexLocation, ri-
>BaseVertexLocation, 0);
}}
```

ShapesApp::Draw

The DrawRenderItems method is invoked in the main Draw call:

```
void ShapesApp::Draw(const GameTimer& gt)
{
    auto cmdListAlloc = mCurrFrameResource->CmdListAlloc;
mCommandList->RSSetViewports(1, &mScreenViewport);
    mCommandList->RSSetScissorRects(1, &mScissorRect);

    // Indicate a state transition on the resource usage.
mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));

    // Clear the back buffer and depth buffer.
    mCommandList->ClearRenderTargetView(CurrentBackBufferView(), Colors::LightSteelBlue, 0,
nullptr);
    mCommandList->ClearDepthStencilView(DepthStencilView(), D3D12_CLEAR_FLAG_DEPTH |
D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);

    // Specify the buffers we are going to render to.
    mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true,
&DepthStencilView());

    ID3D12DescriptorHeap* descriptorHeaps[] = { mCbvHeap.Get() };
    mCommandList->SetDescriptorHeaps(_countof(descriptorHeaps), descriptorHeaps);

mCommandList->SetGraphicsRootSignature(mRootSignature.Get());

    int passCbvIndex = mPassCbvOffset + mCurrFrameResourceIndex;
    auto passCbvHandle = CD3DX12_GPU_DESCRIPTOR_HANDLE(mCbvHeap-
>GetGPUDescriptorHandleForHeapStart());
    passCbvHandle.Offset(passCbvIndex, mCbvSrvUavDescriptorSize);
    mCommandList->SetGraphicsRootDescriptorTable(1, passCbvHandle);

    DrawRenderItems(mCommandList.Get(), mOpaqueRitems);
```

MORE ON ROOT SIGNATURES

A root signature defines what resources need to be bound to the pipeline before issuing a draw call and how those resources get mapped to shader input registers.

What resources need to be bound depends on what resources the current shader programs expect.

When the PSO is created, the root signature and shader programs combination will be validated.

There is a limit of 64 DWORDs (double word: UINT32 bit) that can be put in a root signature

A root signature is defined by an array of root parameters, which can actually be one of three types:

1. *Descriptor Table*: 1 DWORD. References a contiguous range in a heap that identifies the resource to be bound.

2. *Root descriptor (inline descriptor)*: 2 DWORDs. A descriptor is set directly and identifies the resource to be bound; the descriptor does not need to be in a heap. Only CBVs to constant buffers, and SRV/UAVs to buffers can be bound as a root descriptor. In particular, this means SRVs to textures cannot be bound as a root descriptor.

3. *Root constant*: 1 DWORD per 32-bit constant. A list of 32-bit constant values to be bound directly.

```
typedef struct D3D12_ROOT_PARAMETER
    {
    D3D12_ROOT_PARAMETER_TYPE ParameterType;
    union
        {
        D3D12_ROOT_DESCRIPTOR_TABLE DescriptorTable;
        D3D12_ROOT_CONSTANTS Constants;
        D3D12_ROOT_DESCRIPTOR Descriptor;
        } ;
    D3D12_SHADER_VISIBILITY ShaderVisibility;
    } D3D12_ROOT_PARAMETER;
```

ParameterType: A member of the following enumerated type indicating the root parameter type (descriptor table, root constant, CBV root descriptor, SRV root descriptor, UAV root descriptor).

Descriptor Tables

A descriptor table root parameter is further defined by filling out the DescriptorTable member of D3D12_ROOT_PARAMETER.

This simply specifies an array of D3D12_DESCRIPTOR_RANGEs and the number of ranges in the array.

Suppose we defined a table of six descriptors by the following three ranges in order: two CBVs, three SRVs and one UAV. This table would be defined like so:

```
// Create a table with 2 CBVs, 3 SRVs and 1 UAV.
CD3DX12_DESCRIPTOR_RANGE descRange[3];
descRange[0].Init(
D3D12_DESCRIPTOR_RANGE_TYPE_CBV, // descriptor type
2, // descriptor count
0, // base shader register arguments are bound to for this root parameter
0, // register space
0);// offset from start of table

descRange[1].Init(
D3D12_DESCRIPTOR_RANGE_TYPE_SRV, // descriptor type
3, // descriptor count
0, // base shader register arguments are bound to for this root parameter
0, // register space
2);// offset from start of table

descRange[2].Init(
D3D12_DESCRIPTOR_RANGE_TYPE_UAV, // descriptor type
1, // descriptor count
0, // base shader register arguments are bound to for this root parameter
0, // register space
5);// offset from start of table

slotRootParameter[0].InitAsDescriptorTable(
3, descRange, D3D12_SHADER_VISIBILITY_ALL);
```

Root Descriptors & Root Constants

```
typedef struct
D3D12_ROOT_DESCRIPTOR
    {
    UINT ShaderRegister;
    UINT RegisterSpace;
    } D3D12_ROOT_DESCRIPTOR;
```

1. ShaderRegister: The shader register the descriptor will be bound to. For example, if you specify 2 and this root parameter is a CBV then the parameter gets mapped to the constant buffer in register(b2):

cbuffer cbPass : register(b2) {...};

2. RegisterSpace: See D3D12_DESCRIPTOR_RANGE::RegisterSpace.

```
typedef struct D3D12_ROOT_CONSTANTS
    {
    UINT ShaderRegister;
    UINT RegisterSpace;
    UINT Num32BitValues;
    } D3D12_ROOT_CONSTANTS;


// Application code: Root signature definition.
CD3DX12_ROOT_PARAMETER slotRootParameter[1];
slotRootParameter[0].InitAsConstants(12, 0);
// A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(1,
slotRootParameter,
0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
// Application code: to set the constants to register b0.
auto weights = CalcGaussWeights(2.5f);
int blurRadius = (int)weights.size() / 2;
cmdList->SetGraphicsRoot32BitConstants(0, 1,&blurRadius, 0);
cmdList->SetGraphicsRoot32BitConstants(0,(UINT)weights.size(), weights.data(), 1);
```

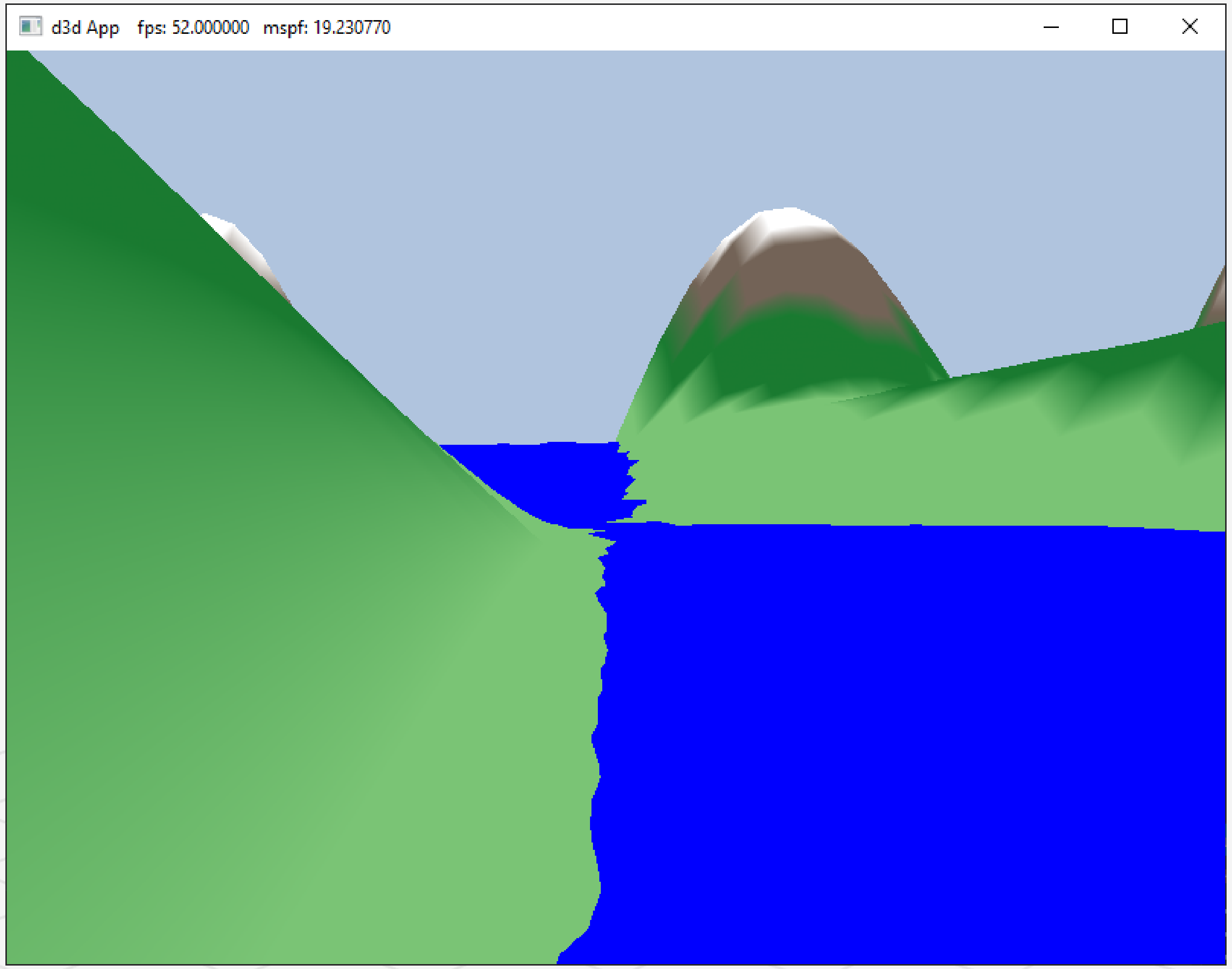
Land and Waves Demo

This demo constructs a triangle grid mesh procedurally and offsets the vertex heights to create a terrain. In addition, it uses another triangle grid to represent water, and animates the vertex heights to create waves.

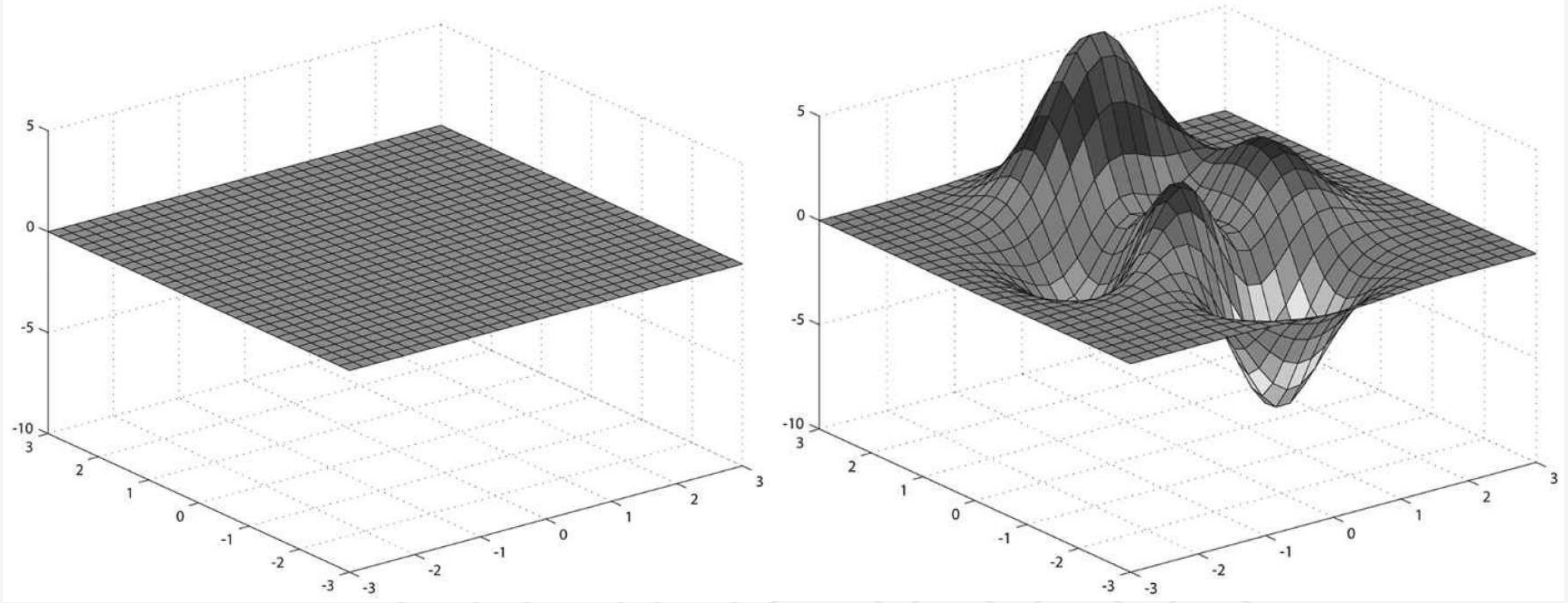
The graph of a "nice" real-valued function $y = f(x, z)$ is a surface. We can approximate the surface by constructing a grid in the *xz*-plane, where every quad is built from two triangles, and then applying the function to each grid point.

The function $f(x, z)$ we have used in this demo is given by:

```
float LandAndWavesApp::GetHeight(float x, float z)const {

return 0.3f*(z*sinf(0.1f*x) + x*cosf(0.1f*z));}
```



Land and Waves Demo



Generating the Grid Vertices

how to build the grid in the *xz*-plane?

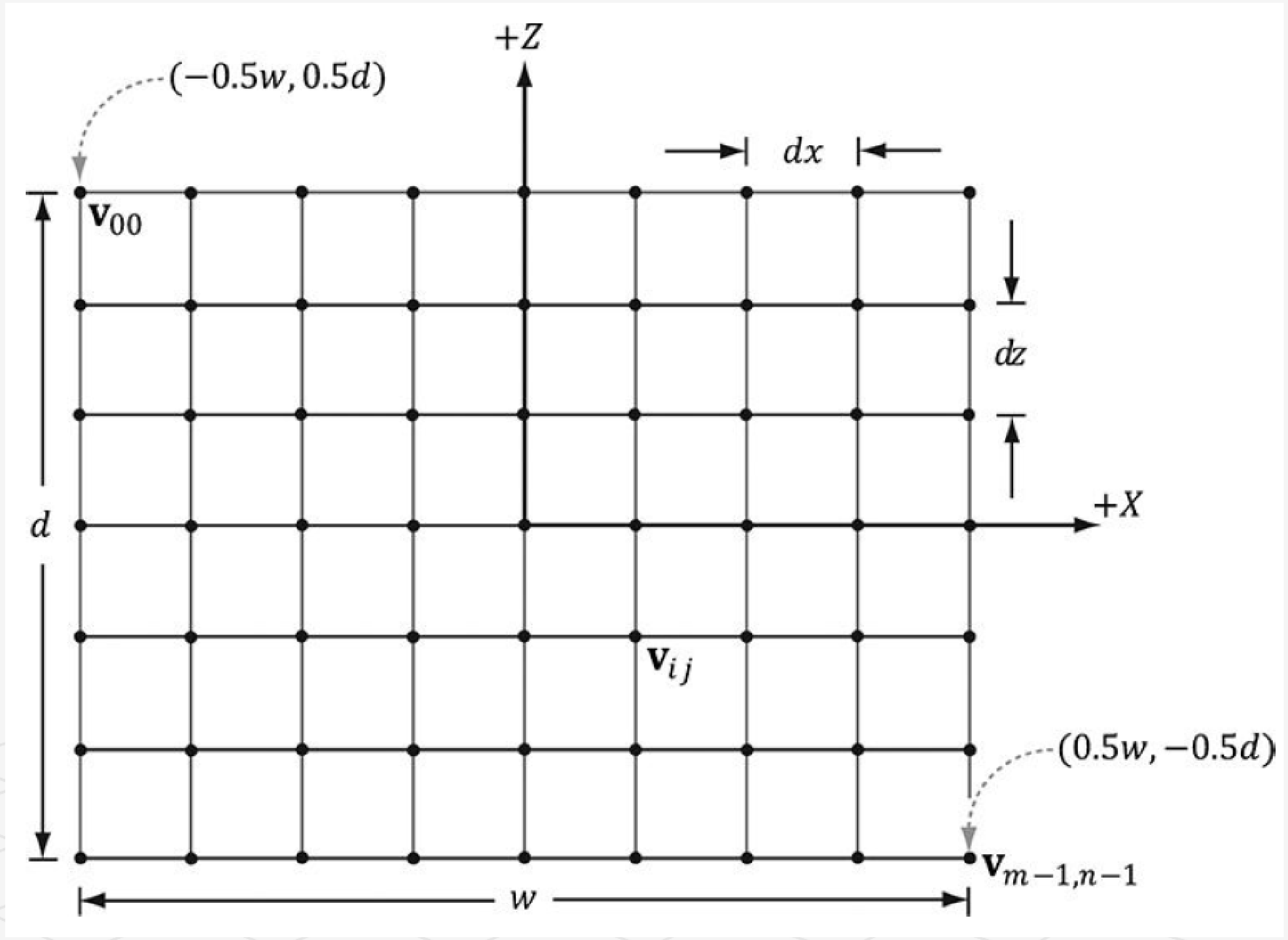
A grid of $m \times n$ vertices induces $(m - 1) \times (n - 1)$ quads (or cells).

Each cell will be covered by two triangles, so there are a total of $2(m - 1) \times (n - 1)$ triangles.

If the grid has width w and depth d, the cell spacing along the *x*-axis is $dx = w/(n - 1)$ and the cell spacing along the *z*-axis is $dz = d/(m - 1)$.

To generate the vertices, we start at the upperleft corner and incrementally compute the vertex coordinates row-by-row. The coordinates of the *ij*th grid vertex in the *xz*-plane are given by:

$$\mathbf{v}_{ij} = [-0.5w + j \cdot dx, \quad 0.0, \quad 0.5d - i \cdot dz]$$



The grid vertices

```
GeometryGenerator::MeshData
GeometryGenerator::CreateGrid(float width,
float depth, uint32 m, uint32 n)
{
    MeshData meshData;

uint32 vertexCount = m*n;
uint32 faceCount   = (m-1)*(n-1)*2;

//
// Create the vertices.
//

float halfWidth = 0.5f*width;
float halfDepth = 0.5f*depth;

float dx = width / (n-1);
float dz = depth / (m-1);

float du = 1.0f / (n-1);
float dv = 1.0f / (m-1);

meshData.Vertices.resize(vertexCount);
```

```
for(uint32 i = 0; i < m; ++i)
{
float z = halfDepth - i*dz;
for(uint32 j = 0; j < n; ++j)
{
float x = -halfWidth + j*dx;

meshData.Vertices[i*n+j].Position = XMFLOAT3(x, 0.0f, z);
meshData.Vertices[i*n+j].Normal   = XMFLOAT3(0.0f, 1.0f, 0.0f);
meshData.Vertices[i*n+j].TangentU = XMFLOAT3(1.0f, 0.0f, 0.0f);

// Stretch texture over grid.
meshData.Vertices[i*n+j].TexC.x = j*du;
meshData.Vertices[i*n+j].TexC.y = i*dv;
}
}
```

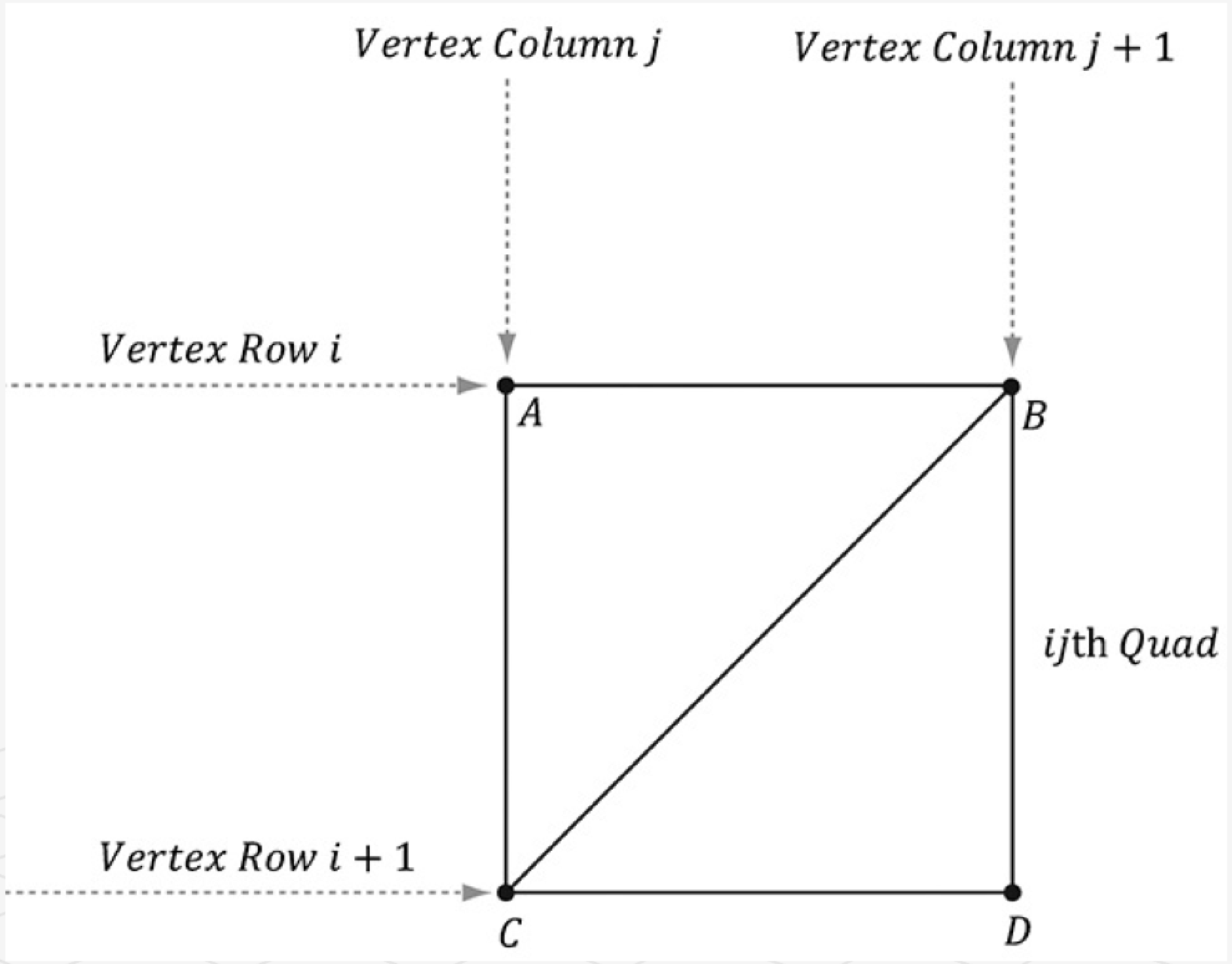
Generating the Grid Indices

We iterate over each quad, again row-by-row starting at the top-left, and compute the indices to define the two triangles of the quad.

For an *m* × *n* vertex grid, the linear array indices of the two triangles are computed as follows:

$$\Delta ABC = \left(i \cdot n + j, \qquad i \cdot n + j + 1, (i+1) \cdot n + j\right)$$

$$\Delta CBD = \left((i+1) \cdot n + j, i \cdot n + j + 1, (i+1) \cdot n + j + 1\right)$$



Grid Indices

```
meshData.Indices32.resize(faceCount*3); // 3 indices per face

// Iterate over each quad and compute indices.
uint32 k = 0;
for(uint32 i = 0; i < m-1; ++i)
{
for(uint32 j = 0; j < n-1; ++j)
{
meshData.Indices32[k]   = i*n+j;
meshData.Indices32[k+1] = i*n+j+1;
meshData.Indices32[k+2] = (i+1)*n+j;

meshData.Indices32[k+3] = (i+1)*n+j;
meshData.Indices32[k+4] = i*n+j+1;
meshData.Indices32[k+5] = (i+1)*n+j+1;

k += 6; // next quad
}
}

    return meshData;
}
```

Applying the Height Function

After we have created the grid, we can extract the vertex elements we want from the `MeshData` grid, turn the flat grid into a surface representing hills, and generate a color for each vertex based on the vertex altitude (*y*-coordinate).

In addition, color the vertices based on their height so we have sandy looking beaches, grassy low hills, and snow mountain peaks.

```
void LandAndWavesApp::BuildLandGeometry()
{
GeometryGenerator geoGen;
GeometryGenerator::MeshData grid = geoGen.CreateGrid(160.0f, 160.0f, 50, 50);

std::vector<Vertex> vertices(grid.Vertices.size());
for(size_t i = 0; i < grid.Vertices.size(); ++i)
{
auto& p = grid.Vertices[i].Position;
vertices[i].Pos = p;
vertices[i].Pos.y = GetHillsHeight(p.x, p.z);

        // Color the vertex based on its height.
        if(vertices[i].Pos.y < -10.0f)
        {
            // Sandy beach color.
            vertices[i].Color = XMFLOAT4(1.0f, 0.96f, 0.62f, 1.0f);
        }
        else if(vertices[i].Pos.y < 5.0f)
        {
            // Light yellow-green.
            vertices[i].Color = XMFLOAT4(0.48f, 0.77f, 0.46f, 1.0f);
        }
        else if(vertices[i].Pos.y < 12.0f)
        {
            // Dark yellow-green.
            vertices[i].Color = XMFLOAT4(0.1f, 0.48f, 0.19f, 1.0f);
        }
        else if(vertices[i].Pos.y < 20.0f)
        {
            // Dark brown.
            vertices[i].Color = XMFLOAT4(0.45f, 0.39f, 0.34f, 1.0f);
        }
        else
        {
            // White snow.
            vertices[i].Color = XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f);
        }
}
```

Root CBVs

Another change we make to the “Land and Waves” demo from the previous “Shape” demos is that we use root descriptors so that we can bind CBVs directly without having to use a descriptor heap.

1. The root signature needs to be changed to take two root CBVs instead of two descriptor tables.

2. No CBV heap is needed nor needs to be populated with descriptors.

3. There is new syntax for binding a root descriptor.

Observe that we use the InitAsConstantBufferView helper method to create root CBV; the parameter specifies the shader register this parameter is bound to.

```
void LandAndWavesApp::BuildRootSignature()
{
    // Root parameter can be a table, root descriptor or root constants.
    CD3DX12_ROOT_PARAMETER slotRootParameter[2];


    // Create root CBV.
    slotRootParameter[0].InitAsConstantBufferView(0);
    slotRootParameter[1].InitAsConstantBufferView(1);


    // A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(2, slotRootParameter, 0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

SetGraphicsRootConstantBufferView

Now, we bind a CBV as an argument to a root descriptor using the following method:

```
virtual void SetGraphicsRootConstantBufferView(
UINT RootParameterIndex,
D3D12_GPU_VIRTUAL_ADDRESS BufferLocation)
```

1. RootParameterIndex: The index of the root parameter we are binding a CBV to.

2. BufferLocation: The virtual address to the resource that contains the constant buffer data.

With this change, our drawing code now looks like this: void LandAndWavesApp::Draw(const GameTimer& gt) →

```
cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);
```

Dynamic Vertex Buffers

A dynamic vertex buffer is where we change the vertex data frequently, say per-frame. For example, suppose we are doing a wave simulation, and we solve the wave equation for the solution function *f*(*x*, *z*, *t*).

Because this function also depends on time *t* (i.e., the wave surface changes with time), we would need to reapply this function to the grid points a short time later (say every 1/30th of a second) to get a smooth animation.

We have already seen an example of uploading data from the CPU to the GPU per frame when we used upload buffers to update our constant buffer data. We can apply the same technique and use our UploadBuffer class, but instead of storing an array of constant buffers, we store an array of vertices:

```
// We cannot update a dynamic vertex buffer until the GPU is done processing
// the commands that reference it.  So each frame needs their own.
    std::unique_ptr<UploadBuffer<Vertex>> WavesVB = nullptr;

WavesVB = std::make_unique<UploadBuffer<Vertex>>(device, waveVertCount, false);


void LandAndWavesApp::UpdateWaves(const GameTimer& gt)
{
// Every quarter second, generate a random wave.
static float t_base = 0.0f;
if((mTimer.TotalTime() - t_base) >= 0.25f)
{
t_base += 0.25f;

int i = MathHelper::Rand(4, mWaves->RowCount() - 5);
int j = MathHelper::Rand(4, mWaves->ColumnCount() - 5);

float r = MathHelper::RandF(0.2f, 0.5f);

mWaves->Disturb(i, j, r);
}

// Update the wave simulation.
mWaves->Update(gt.DeltaTime());

// Update the wave vertex buffer with the new solution.
auto currWavesVB = mCurrFrameResource->WavesVB.get();
for(int i = 0; i < mWaves->VertexCount(); ++i)
{
```