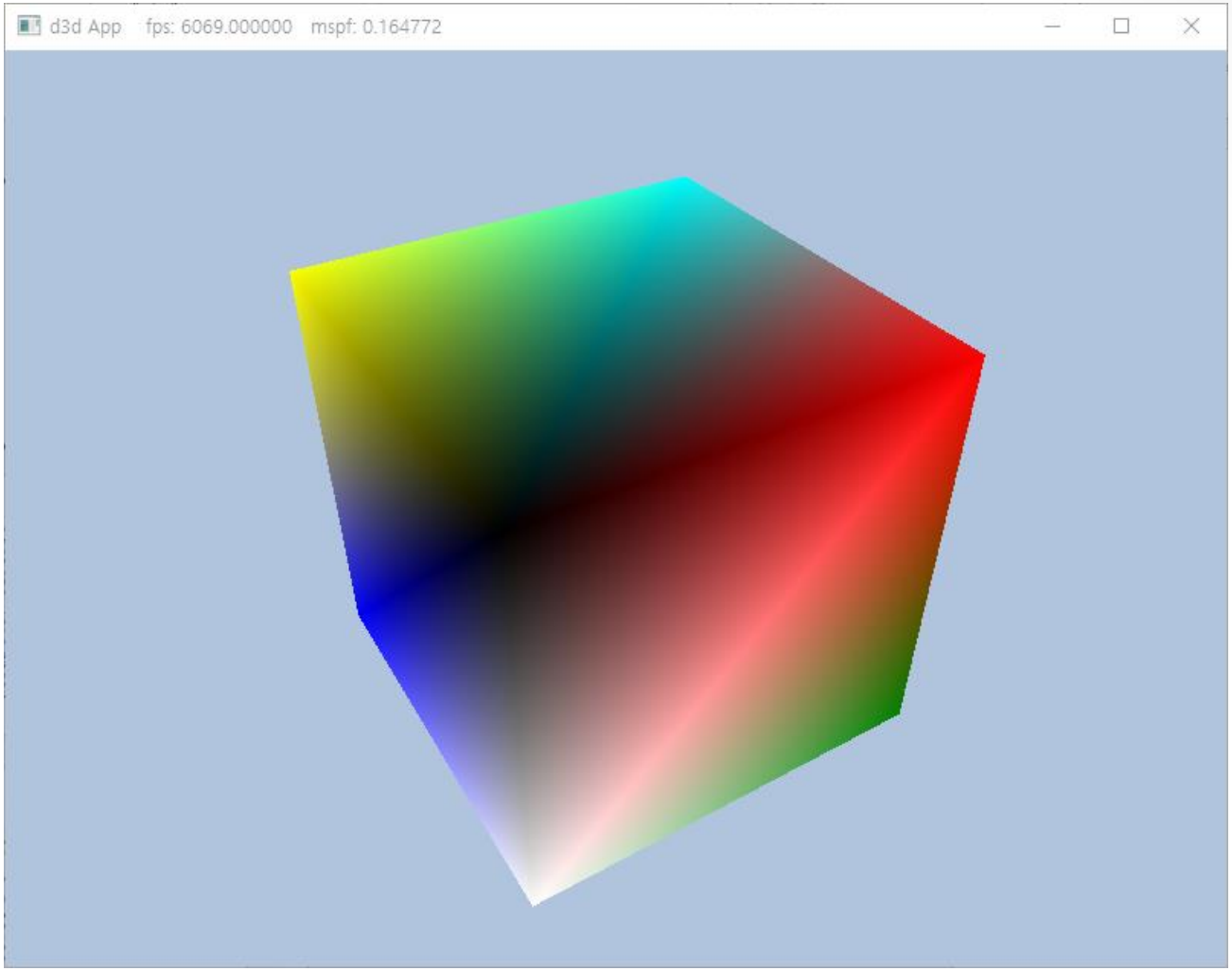
Draw a 3D Box with Solid Coloring



Vertices

- Vertex formats
  - Position & color

```
struct Vertex1 {
  XMFLOAT3 Pos;
  XMFLOAT4 Color;
};
```

  - Position, normal vector & two sets of 2D texture coordinates

```
struct Vertex2 {
  XMFLOAT3 Pos;
  XMFLOAT3 Normal;
  XMFLOAT2 Tex0;
  XMFLOAT2 Tex1;
};
```

Description of Vertex Structure (1)

- Input layout description
  - It describes the input-buffer data for the input-assembler stage.

```
typedef struct D3D12_INPUT_LAYOUT_DESC {
    const D3D12_INPUT_ELEMENT_DESC *pInputElementDescs;
    UINT NumElements;
} D3D12_INPUT_LAYOUT_DESC;
```

- Input element description
  - It describes a single element for the input-assembler stage of the graphics pipeline.

```
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;
```

Description of Vertex Structure (2)

- **SemanticName**
  - The HLSL semantic associated with this element in a shader input-signature.
    - https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-semantics
    - Vertex shader: COLOR[n], NORMAL[n], POSITION[n], TEXCOORD[n], …

```
struct Vertex { XMFLOAT3 Pos;
                XMFLOAT3 Normal;
                XMFLOAT2 TEX0;
                XMFLOAT2 TEX1; }; // vertex structure

// input element description

std::vector<D3D12_INPUT_ELEMENT_DESC> mInputLayout
   ={{"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 },
      {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }};
      {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }};
      {"COLOR", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 32,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0 }};

struct VertexIn { float3 iPos   : POSITION;
                  float3 iNormal : NORMAL;
                  float2 iTex0 : TEXCOORD0;
                  float2 iTex1 : TEXCOORD1; }; // hlsl
```

A **semantic** is a string attached to a shader input or output that conveys information about the intended use of a parameter.

Description of Vertex Structure (3)

- **`SemanticIndex`**
  - The semantic index for the element.
- **`Format`**
  - A **`DXGI_FORMAT`**-typed value that specifies the format of the element data.
  - https://learn.microsoft.com/en-us/windows/win32/api/dxgiformat/ne-dxgiformat-dxgi_format
- **`InputSlot`**
  - An integer value that identifies the input-assembler. Valid values are between 0 and 15.
- **`AlignedByteOffset`**
  - Optional. Offset, in bytes, to this element from the start of the vertex.
- **`InputSlotClass`**
  - A value that identifies the input data class for a single input slot.
- **`InstanceDataStepRate`**
  - The number of instances to draw using the same per-instance data before advancing in the buffer by one element. This value must be 0 for an element that contains per-vertex data (the slot class is set to the **`D3D12_INPUT_PER_VERTEX_DATA`** member of **`D3D12_INPUT_CLASSIFICATION`**).

Vertex Buffers (1)

- Vertex buffer
  - In order for the GPU to access an array of vertices, they need to be placed in a GPU resource (**ID3D12Resource**) called a buffer. We call a buffer that stores vertices a vertex buffer.
  - You can create an **ID3D12Resource** object by filling out a **D3D12_RESOURCE_DESC** structure describing the buffer resource, and then calling the **ID3D12Device::CreateCommittedResource** method.

```
// d3dUtil.cpp
Microsoft::WRL::ComPtr<ID3D12Resource>
d3dUtil::CreateDefaultBuffer(
    ID3D12Device* device,
    ID3D12GraphicsCommandList* cmdList,
    const void* initData,
    UINT64 byteSize,
    Microsoft::WRL::ComPtr<ID3D12Resource>& uploadBuffer)
{ /*… CreateCommittedResource …*/ }
```

```
// Application.cpp
// std::unique_ptr<MeshGeometry> mBoxGeo = nullptr;
mBoxGeo->VertexBufferGPU =
    d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
        mCommandList.Get(), vertices.data(), vbByteSize,
        mBoxGeo->VertexBufferUploader);
mBoxGeo->IndexBufferGPU =
    d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
        mCommandList.Get(), indices.data(), ibByteSize,
        mBoxGeo->IndexBufferUploader);
```

Vertex Buffers (2)

```
Microsoft::WRL::ComPtr<ID3D12Resource> d3dUtil::CreateDefaultBuffer( /* … */ ) {
    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())));
// …
}
```

```
struct CD3DX12_RESOURCE_DESC  : public
D3D12_RESOURCE_DESC{
  CD3DX12_RESOURCE_DESC();
  explicit CD3DX12_RESOURCE_DESC(/* … */);
  CD3DX12_RESOURCE_DESC (/* … */);
  CD3DX12_RESOURCE_DESC static inline Buffer(/* … */);
  CD3DX12_RESOURCE_DESC static inline Buffer(/* … */);
  CD3DX12_RESOURCE_DESC static inline Tex1D(/* … */);
  CD3DX12_RESOURCE_DESC static inline Tex2D(/* … */);
  CD3DX12_RESOURCE_DESC static inline Tex3D(/* … */);
  // …
};
```

Vertex/Index Buffer View (1)

- Vertex buffer view

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

- Index buffer view

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

Vertex/Index Buffer View (2)

- After a vertex buffer has been created and we have created a view to it, we can bind it to an input slot of the pipeline to feed the vertices to the input assembler stage of the pipeline. This can be done with the following method:

```
void ID3D12GraphicsCommandList::IASetVertexBuffers(
 UINT StartSlot,
 UINT NumBuffers,
 const D3D12_VERTEX_BUFFER_VIEW *pViews);
```

  - **`StartSlot`**: The input slot to start binding vertex buffers to. There are 16 input slots indexed from 0-15.
  - **`NumBuffers`**: The number of vertex buffers we are binding to the input slots. If the start slot has index k and we are binding n buffers, then we are binding buffers to input slots I_k, I_k+1,...,I_k+n-1.
  - **`pViews`**: Pointer to the first element of an array of vertex buffers views.

Vertex/Index Buffer View (3)

- Example)

```
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);
```

Vertex/Index Buffer View (4)

- 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 ID3D12CommandList::DrawInstanced(
    UINT VertexCountPerInstance,
    UINT InstanceCount,
    UINT StartVertexLocation,
    UINT StartInstanceLocation);
```

  - **`VertexCountPerInstance`**: The number of vertices to draw (per instance).
  - **`InstanceCount`**: Used for an advanced technique called instancing; for now, set this to 1 as we only draw one instance.
  - **`StartVertexLocation`**: specifies the index (zero-based) of the first vertex in the vertex buffer to begin drawing.
  - **`StartInstanceLocation`**: Used for an advanced technique called instancing; for now, set this to 0.

Vertex/Index Buffer View (5)

- Vertex
  - **IASetVertexBuffers → DrawInstanced**
- Index
  - **IASetVertexBuffers → IASetIndexBuffer → DrawIndexedInstanced**

- **ID3D12GraphicsCommandList::IASetIndexBuffer**

```
void IASetIndexBuffer(
  [in, optional] const D3D12_INDEX_BUFFER_VIEW *pView
);
```

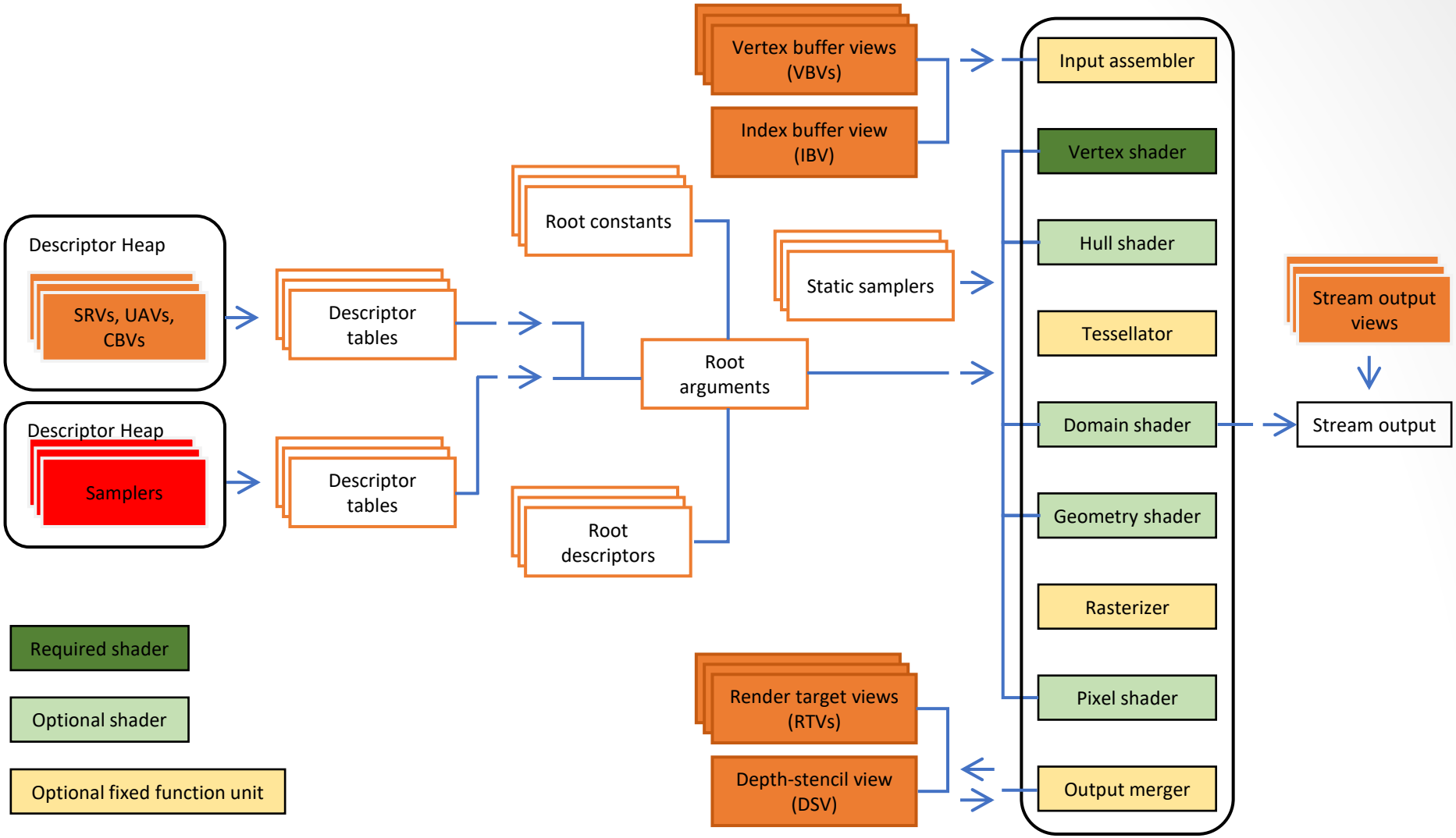
Vertex/Index Buffer View (6)

- **ID3D12GraphicsCommandList::DrawIndexedInstanced**

```
void DrawIndexedInstanced(
  [in] UINT IndexCountPerInstance,
  [in] UINT InstanceCount,
  [in] UINT StartIndexLocation,
  [in] INT  BaseVertexLocation,
  [in] UINT StartInstanceLocation
);
```

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

Direct3D 12 Graphics Pipeline



HLSL (1)

- HLSL (high level shading language)
  - Programming language for GPU in DirectX

```
// color.hlsl
cbuffer cbPerObject : register(b0) {
    float4x4 gWorldViewProj;
};
struct VertexIn {
    float3 PosL  : POSITION;
    float4 Color : COLOR;
};
struct VertexOut {
    float4 PosH  : SV_POSITION;
    float4 Color : COLOR;
};
VertexOut VS(VertexIn vin) {  // vertex shader
    VertexOut vout;
    // Transform to homogeneous clip space.
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);
    // Just pass vertex color into the pixel shader.
    vout.Color = vin.Color;
    return vout;
}
float4 PS(VertexOut pin) : SV_Target {
    return pin.Color;
}
```


Kyung Hee University
nize@khu.ac.kr

HLSL (2)

```
// equivalent code
cbuffer cbPerObject : register(b0) {
   float4x4 gWorldViewProj;
};

void VS(float3 iPos : POSITION, float4 iColor : COLOR,
   out float4 oPos : SV_POSITION, out float4 oColor : COLOR) {
    oPos = mul(float4(iPos, 1.0f), gWorldViewProj);
    oColor = iColor;
}

float4 PS(float4 oPos : SV_POSITION, float4 oColor : COLOR)
   : SV_Target {
    return oColor;
}
```


Kyung Hee University
nize@khu.ac.kr

Vertex Shader (1)

```
// color.hlsl
void VS(float3 iPos : POSITION, float4 iColor : COLOR,
   out float4 oPos : SV_POSITION, out float4 oColor : COLOR) {
    oPos = mul(float4(iPos, 1.0f), gWorldViewProj);
    oColor = iColor;
} // SV stands for system value

// C++ code
std::vector<D3D12_INPUT_ELEMENT_DESC> InputLayout = {
   {"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 }
};
struct Vertex {
   XMFLOAT3 Pos;
   XMFLOAT4 Color;
};
```

Vertex Shader (2)

- Output position of vertex shader
  - In homogeneous space.
  - For Direct3D 10 and later, use SV_POSITION.
- `oPos = mul(float4(iPos, 1.0f), gWorldViewProj);`
  - `float4(iPos, 1.0f)` → w=1
  - **pMVP**
-
```
cbuffer cbPerObject : register(b0) {
  float4x4 gWorldViewProj;
};
```
  - **cbuffer** (constant buffer) type object: **cbPerObject**
  -
```
// C++ code
ObjectConstants objConstants;
XMStoreFloat4x4(&objConstants.WorldViewProj,
XMMatrixTranspose(worldViewProj));
mObjectCB->CopyData(0, objConstants);
```
  - **register**: Optional keyword for assigning a shader variable to a particular register. (b: constant buffer, t: texture and texture buffer, c: buffer offset, …, #: register number)

XMMatrixTranspose: HLSL by default expects column-major packed matrix.

Pixel Shader

```
// color.hlsl
VertexOut VS(VertexIn vin) {
    VertexOut vout;
    vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);
    vout.Color = vin.Color;
    return vout;
}
float4 PS(VertexOut pin) : SV_Target {
    return pin.Color;
}
// Each pixel shader input parameter has an attached semantics.
// These semantics are used to map vertex shader outputs into
// the pixel shader input parameters.

// SV_Target: semantic for the return type
// (SV_TARGET: COLOR(Direct3D 9)
// semantics in HLSL are case insensitive
```

HLSL Reference (1)

- Scalar types
  - **`bool, int, half`** (half precision floating point)**`, float, double`**
- Vector types
  - **`float2, float3, float4`**
  - **`v[i], v.x, v.y,`** …
  - Swizzling: copy any source register component to any temporary register component.
    - **`r.[xyzw][xyzw][xyzw][xyzw]`**
      - e.g., **`r.wyyx, r.wzyx, r.xy,`** …
- Matrix types
  - float2x2, float3x3, float4x4, float 3x4
  - **`m[i][j]`**
- Arrays
- Structures

HLSL Reference (2)

- **`typedef`** keyword
- Casting
- **`if, else`**
- **`for, while, do … while`**
- Function parameter
  - **`in`** (default), **`out, inout`**
- Built-in functions
  - **`Abe, ceil, cos, floor, log, log10, log2, max, min, sin, tan, sqrt`**
  - **`Clamp(x, ab, b), cross(u, v), ddx(p), ddy(p), degrees(x), determinant(M), distance(u, v), dot(u, v), frac(x), length(v), lerp(u, v, t), mul(M, N), normalize(v), radians(x), saturate(x), sincos(in x, out s, out c), reflect(v, n), refract(v, n, eta), rsqrt(x)`**

Creating Constant Buffers (1)

- 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.
  - For example, if the camera is moving every frame, the constant buffer would need to be updated with the new view matrix every frame.
- Constant buffers also have the special hardware requirement that their size must be a multiple of the minimum hardware allocation size (256 bytes).
  - ```
    // d3dUtil.h
    class d3dUtil{
    // …
       static UINT CalcConstantBufferByteSize(UINT byteSize)
       {   return (byteSize + 255) & ~255;    }
    };
    ```
    ```

Creating Constant Buffers (2)

- ```
  // MathHelper.h
  class MathHelper {
  // …
     static DirectX::XMFLOAT4X4 Identity4x4()     {
          static DirectX::XMFLOAT4X4 I(
              1.0f, 0.0f, 0.0f, 0.0f,
              0.0f, 1.0f, 0.0f, 0.0f,
              0.0f, 0.0f, 1.0f, 0.0f,
              0.0f, 0.0f, 0.0f, 1.0f);
          return I;
      }
  };
  ```
- ```
  // UploadBuffer.h
  template<typename T>
  class UploadBuffer{
  public:
      UploadBuffer(ID3D12Device* device, UINT elementCount,
             bool isConstantBuffer)
     : mIsConstantBuffer(isConstantBuffer) { /* … */ }
  // …
  };
  ```

Creating Constant Buffers (3)

```
// Application.cpp
struct ObjectConstants{
    XMFLOAT4X4 WorldViewProj = MathHelper::Identity4x4();
}

std::unique_ptr<UploadBuffer<ObjectConstants>> mObjectCB = nullptr;

void Applicstion::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 & Descriptor Tables (1)

- Generally, different shader programs will expect different resources to be bound to the rendering pipeline before a draw call is executed.
- Resources are bound to particular register slots, where they can be accessed by shader programs.
- The **root signature** defines what types of resources are bound to the graphics pipeline.

In programming, a signature (also known as a type signature or method signature) describes the input and output of a function or method, including the function name, parameters, and their types, and the return type.
Root signatures are a complex data structure containing nested structures (direct3d12).

Root Signature & Descriptor Tables (2)

```
// Texture resource bound to texture register slot 0.
Texture2D  gDiffuseMap : register(t0);

// Sampler resources bound to sampler register slots 0-5.
SamplerState gsamPointWrap        : register(s0);
SamplerState gsamPointClamp       : register(s1);
SamplerState gsamLinearWrap       : register(s2);
SamplerState gsamLinearClamp      : register(s3);
SamplerState gsamAnisotropicWrap  : register(s4);
SamplerState gsamAnisotropicClamp : register(s5);
```

Root Signature & Descriptor Tables (3)

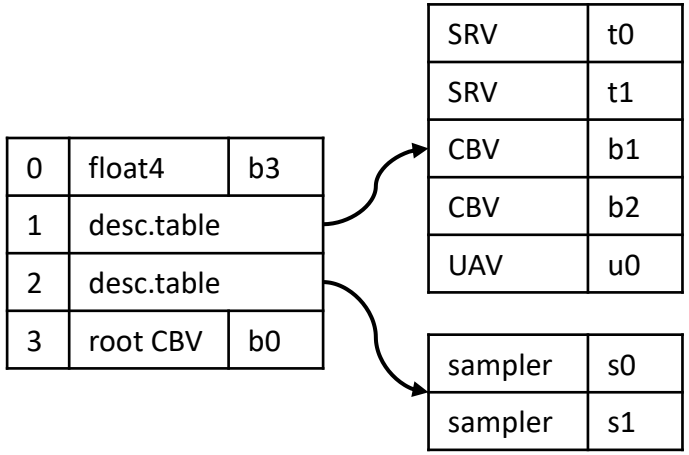
```
// cbuffer resource bound to cbuffer register slots 0-2
cbuffer cbPerObject : register(b0){
  float4x4 gWorld;
  float4x4 gTexTransform;
};

// Constant data that varies per material.
cbuffer cbPass : register(b1) {
  float4x4 gView;
  float4x4 gProj;
  // …
};

cbuffer cbMaterial : register(b2) {
  float4  gDiffuseAlbedo;
  float3  gFresnelR0;
  float  gRoughness;
  float4x4 gMatTransform;
};
```

Root Signature & Descriptor Tables (4)

- A root signature is represented in Direct3D by the **`ID3D12RootSignature`** interface.
  - It is defined by an array of root parameters that describe the resources the shaders expect for a draw call.
  - A root parameter can be a root constant, root descriptor, or descriptor table.
  - A descriptor table specifies a contiguous range of descriptors in a descriptor heap.



We will discuss root constants and root descriptors in the next chapter; in this chapter, we will just use descriptor tables.

Root Signature & Descriptor Tables (5)

- **D3D12_ROOT_PARAMETER** structure describes the slot of a root signature version 1.0.
- **CD3DX12_ROOT_PARAMETER** structure is a helper structure to enable easy initialization of a **D3D12_ROOT_PARAMETER** structure.

```
struct CD3DX12_ROOT_PARAMETER  : public D3D12_ROOT_PARAMETER{
// …
};
```

Helper structures and functions for Direct3D 12
- These helper structures and helper functions are declared in d3dx12.h. d3dx12.h is available separately from the Direct3D 12 headers and is not included in the Windows SDK (It is available for download from Microsoft, https://github.com/microsoft/DirectX-Graphics-Samples/blob/master/Libraries/D3D12RaytracingFallback/Include/d3dx12.h).
- Each helper structure has a 'C' prefix and is associated with a D3D12 structure which lacks the 'C' prefix.

Root Signature & Descriptor Tables (6)

```
// Shader programs typically require resources as input (constant buffers,
// textures, samplers).  The root signature defines the resources the shader
// programs expect.  If we think of the shader programs as a function, and
// the input resources as function parameters, then the root signature can be
// thought of as defining the function signature.

// 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);
```

Root Signature & Descriptor Tables (7)

```
// 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)));
```

Root Signature & Descriptor Tables (8)

- 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, you can use the ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable to bind a descriptor table to the pipeline:

```
void SetGraphicsRootDescriptorTable(
  [in] UINT                        RootParameterIndex,
  [in] D3D12_GPU_DESCRIPTOR_HANDLE BaseDescriptor
);
```

  - **`RootParameterIndex`**: Index of the root parameter you are setting.
  - **`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.

Root Signature & Descriptor Tables (9)

```
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 Shader (1)

- Compiling HLSL code into bytecode for a given target.
  - ```
    HRESULT D3DCompileFromFile(
      [in]             LPCWSTR filename,
      [in, optional]   const D3D_SHADER_MACRO *pDefines,
      [in, optional]   ID3DInclude            *pInclude,
      [in]             LPCSTR                 pEntrypoint,
      [in]             LPCSTR                 pTarget,
      [in]             UINT                   Flags1,
      [in]             UINT                   Flags2,
      [out]            ID3DBlob               **ppCode,
      [out, optional]  ID3DBlob               **ppErrorMsgs
    );
    ```
    - [in] pFileName: Shader code.
    - [in, optional] pDefines: Defining shader macros.
    - [in, optional] pInclude: Handling include files.
    - [in] pEntrypoint: Shader entry point function where shader execution begins.
    - [in] pTarget: Shader target or set of shader features to compile against.
      - vs_5_0/vs_5_1: Vertex shader 5.0/5.1, hs_5_0/hs_5_1: Hull shader, ds_5_0 / ds_5_1: Domain shader, gs_5_0 / gs_5_1: Geometry shader, ps_5_0 / ps_5_1: Pixel shader, cs_5_0 / cs_5_1: Compute shader
    - [in] Flags1: Shader compile options
    - [in] Flags2: Effect compile options
    - [out] ppCode: Accessing the compiled code.
    - [out, optional] ppErrorMsgs: Error messages
    ```

Compiling Shader (2)

- `// d3dUtil.cpp`

```
ComPtr<ID3DBlob> d3dUtil::CompileShader(
  const std::wstring& filename,
  const D3D_SHADER_MACRO* defines,
  const std::string& entrypoint,
  const std::string& target) {
  UINT compileFlags = 0;

#if defined(DEBUG) || defined(_DEBUG)
  compileFlags = D3DCOMPILE_DEBUG | D3DCOMPILE_SKIP_OPTIMIZATION;
#endif
  HRESULT hr = S_OK;

  ComPtr<ID3DBlob> byteCode = nullptr;
  ComPtr<ID3DBlob> errors;
  hr = D3DCompileFromFile(filename.c_str(), defines,
    D3D_COMPILE_STANDARD_FILE_INCLUDE, entrypoint.c_str(),
    target.c_str(), compileFlags, 0, &byteCode, &errors);
  if(errors != nullptr)
     OutputDebugStringA((char*)errors->GetBufferPointer());
  ThrowIfFailed(hr);

  return byteCode;
}
```

Offline Compilation (1)

- Compiled shader object (.cso)
- To compile shaders offline we use the **FXC** tool that comes with DirectX.
  - This is a command line tool.
  - 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”

Run: Developer Command Prompt for VS2022

Offline Compilation (2)

- /Od: Disable optimizations. /Od implies /Gfp, though output may not be identical to /Od /Gfp.
- /O0 /O1, /O2, /O3: Optimization levels. O1 is the default setting.
  - O0 - Disables instruction reordering.
  - O1 - Disables instruction reordering for ps_3_0 and up.
  - O2 - Same as O1. Reserved for future use.
  - O3 - Same as O1. Reserved for future use.
- /Zi: Enable debugging information.
- /T <profile>: Shader type and target version.
- /E <name>: Shader entry point.
- /Fo <file>: Output object file
- /Fc <file>: Output assembly code listing file.
- https://learn.microsoft.com/en-us/windows/win32/direct3dtools/dx-graphics-tools-fxc-syntax

Offline Compilation (3)

- // d3dUtil.cpp

```
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);

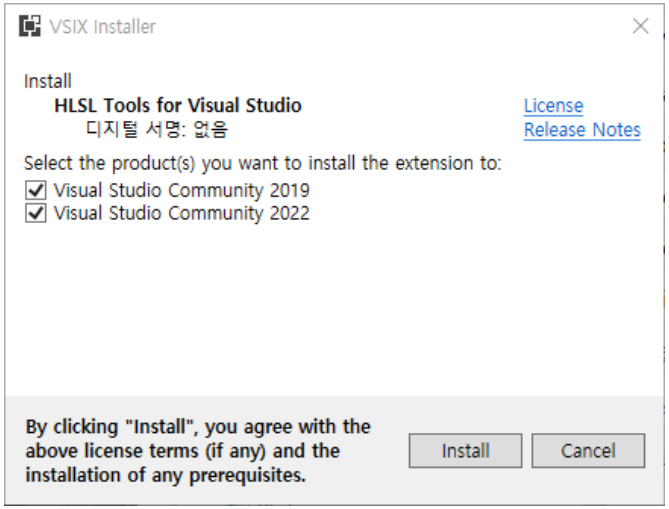
    ComPtr<ID3DBlob> blob;
    ThrowIfFailed(D3DCreateBlob(size,
blob.GetAddressOf()));

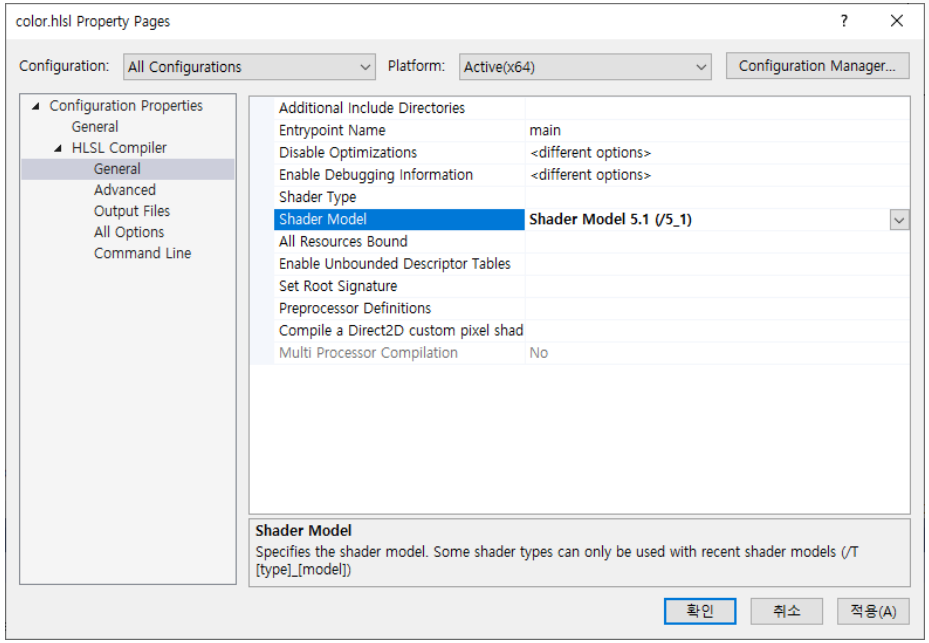
    fin.read((char*)blob->GetBufferPointer(), size);
    fin.close();

    return blob;
}
```

HLSL Tools

- Install HLSL Tools
  - https://marketplace.visualstudio.com/items?itemName=TimGJones.HLSLToolsforVisualStudio





Rasterizer State (1)

- Rasterizer state description
  - ```
    typedef struct D3D12_RASTERIZER_DESC {
      D3D12_FILL_MODE                    FillMode;
      D3D12_CULL_MODE                    CullMode;
      BOOL                               FrontCounterClockwise;
      INT                                DepthBias;
      FLOAT                              DepthBiasClamp;
      FLOAT                              SlopeScaledDepthBias;
      BOOL                               DepthClipEnable;
      BOOL                               MultisampleEnable;
      BOOL                               AntialiasedLineEnable;
      UINT                               ForcedSampleCount;
      D3D12_CONSERVATIVE_RASTERIZATION_MODE
                                         ConservativeRaster;
    } D3D12_RASTERIZER_DESC;
    ```
    ```

Rasterizer State (2)

- `// Application.cpp`

```
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.RasterizerState.FillMode = D3D12_FILL_MODE_WIREFRAME;
// …
}
```

Pipeline State Object

- Pipeline state object (PSO)
  - Most of the objects that control the state of the graphics pipeline are specified as an aggregate called a pipeline state object (PSO).

```
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;
```

Geometry Helper Structure (1)

- It is helpful to create a structure that groups a vertex and index buffer together to define a group of geometry.
- In addition, this structure can keep a system memory backing of the vertex and index data so that it can be read by the CPU.
- The CPU will need access to the geometry data for things like picking and collision detection.

Geometry Helper Structure (2)

```
// d3dUtil.h
struct SubmeshGeometry  {
    UINT IndexCount = 0;
    UINT StartIndexLocation = 0;
    INT BaseVertexLocation = 0;
    DirectX::BoundingBox Bounds;
};
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;

    // A MeshGeometry may store multiple geometries in one vertex/index buffer.
    // Use this container to define the Submesh geometries so we can draw
    // the Submeshes individually.
    std::unordered_map<std::string, SubmeshGeometry> DrawArgs;
```


Kyung Hee University
nize@khu.ac.kr

Geometry Helper Structure (3)

```
    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;
    }

    // We can free this memory after we finish upload to the GPU.
    void DisposeUploaders() {
        VertexBufferUploader = nullptr;
        IndexBufferUploader = nullptr;
    }
};
```

Geometry Helper Structure (4)

```
std::unique_ptr<MeshGeometry> mBoxGeo = nullptr;


mBoxGeo = std::make_unique<MeshGeometry>();
mBoxGeo->Name = "boxGeo";
// …

mBoxGeo->VertexByteStride = sizeof(Vertex);
// …

SubmeshGeometry submesh;
submesh.IndexCount = (UINT)indices.size();
// …

mBoxGeo->DrawArgs["box"] = submesh;
```

Geometry Helper Structure (5)

```
std::unordered_map<std::string, std::unique_ptr<MeshGeometry>>
mGeometries;

SubmeshGeometry boxSubmesh;
// …
SubmeshGeometry gridSubmesh;
// …
SubmeshGeometry sphereSubmesh;
// …
SubmeshGeometry cylinderSubmesh;
// …

auto geo = std::make_unique<MeshGeometry>();
geo->Name = "shapeGeo";
// …

mGeometries[geo->Name] = std::move(geo);
```

BoxApp Class - Declaration (1)

```
// BoxApp.cpp
struct Vertex {
    XMFLOAT3 Pos;
    XMFLOAT4 Color;
};
struct ObjectConstants {
    XMFLOAT4X4 WorldViewProj = MathHelper::Identity4x4();
};
class BoxApp : public D3DApp {
public:
    BoxApp(HINSTANCE hInstance);
    BoxApp(const BoxApp& rhs) = delete;
    BoxApp& operator=(const BoxApp& rhs) = delete;
    ~BoxApp();
   virtual bool Initialize()override;
   // heaps, buffers, root signature, shader, input, layout, mesh, PSO, …
private:
    virtual void OnResize()override;
    virtual void Update(const GameTimer& gt)override;        // WVP
    virtual void Draw(const GameTimer& gt)override;          // Rendering

    virtual void OnMouseDown(WPARAM btnState, int x, int y)override;
    virtual void OnMouseUp(WPARAM btnState, int x, int y)override;
    virtual void OnMouseMove(WPARAM btnState, int x, int y)override;
```

BoxApp Class - Declaration (2)

```
// BoxApp.cpp
    void BuildDescriptorHeaps();
    void BuildConstantBuffers();
    void BuildRootSignature();
    void BuildShadersAndInputLayout();
    void BuildBoxGeometry();
    void BuildPSO();

private:
    ComPtr<ID3D12RootSignature> mRootSignature = nullptr;
    ComPtr<ID3D12DescriptorHeap> mCbvHeap = nullptr;

    std::unique_ptr<UploadBuffer<ObjectConstants>> mObjectCB = nullptr;

    std::unique_ptr<MeshGeometry> mBoxGeo = nullptr;

    ComPtr<ID3DBlob> mvsByteCode = nullptr;
    ComPtr<ID3DBlob> mpsByteCode = nullptr;
```

BoxApp Class - Declaration (3)

```
// BoxApp.cpp
    std::vector<D3D12_INPUT_ELEMENT_DESC> mInputLayout;

    ComPtr<ID3D12PipelineState> mPSO = nullptr;

    XMFLOAT4X4 mWorld = MathHelper::Identity4x4();
    XMFLOAT4X4 mView = MathHelper::Identity4x4();
    XMFLOAT4X4 mProj = MathHelper::Identity4x4();

    float mTheta = 1.5f*XM_PI;
    float mPhi = XM_PIDIV4;
    float mRadius = 5.0f;

    POINT mLastMousePos;
};
```

WinMain

```
// BoxApp.cpp
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE prevInstance,
    PSTR cmdLine, int showCmd) {
   // Enable run-time memory check for debug builds.
#if defined(DEBUG) | defined(_DEBUG)
   _CrtSetDbgFlag( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
#endif
    try {
        BoxApp theApp(hInstance);
        if(!theApp.Initialize())
            return 0;

        return theApp.Run();
    }
    catch(DxException& e) {
        MessageBox(nullptr, e.ToString().c_str(), L"HR Failed", MB_OK);
        return 0;
    }
}
```

BoxApp Class - Definition (1)

```
// BoxApp.cpp
BoxApp::BoxApp(HINSTANCE hInstance) : D3DApp(hInstance) {
}
BoxApp::~BoxApp() {
}
bool BoxApp::Initialize() {
    if(!D3DApp::Initialize()) return false;
    // Reset the command list to prep for initialization commands.
    ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(), nullptr));

    BuildDescriptorHeaps();              // Descriptor Heaps
    BuildConstantBuffers();              // Constant Buffers
    BuildRootSignature();                // Root Signature
    BuildShadersAndInputLayout();        // Compile Shader & Layout
    BuildBoxGeometry();                  // Object Mesh
    BuildPSO();                          // Pipeline State Object

    // Execute the initialization commands.
    ThrowIfFailed(mCommandList->Close());
    ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
    mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

    // Wait until initialization is complete.
    FlushCommandQueue();
    return true;
}
```

BoxApp Class - Definition (2)

```
// BoxApp.cpp
void BoxApp::OnResize() {
    D3DApp::OnResize();
    // The window resized, so update the aspect ratio and recompute the projection matrix.
    XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi, AspectRatio(), 1.0f, 1000.0f);
    XMStoreFloat4x4(&mProj, P);
}
void BoxApp::Update(const GameTimer& gt) { // World, View & Projection Matrices
    // 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);
}
```

BoxApp Class - Definition (3)

```
// BoxApp.cpp
void BoxApp::Draw(const GameTimer& gt) {
    // Reuse the memory associated with command recording.
    // We can only reset when the associated command lists have finished execution on the GPU.
    ThrowIfFailed(mDirectCmdListAlloc->Reset());

    // A command list can be reset after it has been added to the command queue via ExecuteCommandList.
    // Reusing the command list reuses memory.
    ThrowIfFailed(mCommandList->Reset(mDirectCmdListAlloc.Get(), mPSO.Get()));

    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);
```

BoxApp Class - Definition (4)

```
// BoxApp.cpp

    // 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());

    mCommandList->IASetVertexBuffers(0, 1, &mBoxGeo->VertexBufferView());
    mCommandList->IASetIndexBuffer(&mBoxGeo->IndexBufferView());
```

BoxApp Class - Definition (5)

```
// BoxApp.cpp
    mCommandList->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
    mCommandList->SetGraphicsRootDescriptorTable(0,
        mCbvHeap->GetGPUDescriptorHandleForHeapStart());
    mCommandList->DrawIndexedInstanced( mBoxGeo->DrawArgs["box"].IndexCount,
        1, 0, 0, 0);

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

    // Done recording commands.
    ThrowIfFailed(mCommandList->Close());

    // Add the command list to the queue for execution.
    ID3D12CommandList* cmdsLists[] = { mCommandList.Get() };
    mCommandQueue->ExecuteCommandLists(_countof(cmdsLists), cmdsLists);

    // swap the back and front buffers
    ThrowIfFailed(mSwapChain->Present(0, 0));
    mCurrBackBuffer = (mCurrBackBuffer + 1) % SwapChainBufferCount;

    // Wait until frame commands are complete.  This waiting is inefficient and is
    // done for simplicity.  Later we will show how to organize our rendering code
    // so we do not have to wait per frame.
    FlushCommandQueue();
}
```

BoxApp Class - Definition (6)

```
// BoxApp.cpp
void BoxApp::OnMouseDown(WPARAM btnState, int x, int y) {
    mLastMousePos.x = x;      mLastMousePos.y = y;
    SetCapture(mhMainWnd);
}
void BoxApp::OnMouseUp(WPARAM btnState, int x, int y) { ReleaseCapture(); }
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 Class - Definition (7)

```
// BoxApp.cpp
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)));
}
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());
}
```

BoxApp Class - Definition (8)

```
// BoxApp.cpp
void BoxApp::BuildRootSignature() {
    // Shader programs typically require resources as input (constant buffers,
    // textures, samplers).  The root signature defines the resources the shader
    // programs expect.  If we think of the shader programs as a function, and
    // the input resources as function parameters, then the root signature can be
    // thought of as defining the function signature.

    // 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);
```

BoxApp Class - Definition (9)

```
// BoxApp.cpp
    // 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)));
}
```

BoxApp Class - Definition (10)

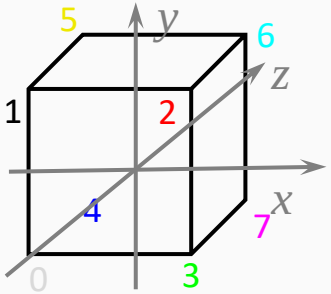
```
// BoxApp.cpp
void BoxApp::BuildShadersAndInputLayout() {
    HRESULT hr = S_OK;

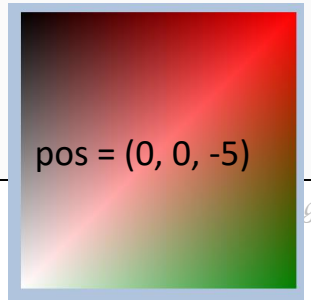
   mvsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
       "VS", "vs_5_0");
   mpsByteCode = d3dUtil::CompileShader(L"Shaders\\color.hlsl", nullptr,
       "PS", "ps_5_0");

    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 }
    };
}
```

BoxApp Class - Definition (11)

```
// BoxApp.cpp
void BoxApp::BuildBoxGeometry() {
    std::array<Vertex, 8> vertices = {
       Vertex({ XMFLOAT3(-1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::White) }), // 0
       Vertex({ XMFLOAT3(-1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Black) }), // 1
       Vertex({ XMFLOAT3(+1.0f, +1.0f, -1.0f), XMFLOAT4(Colors::Red) }),   // 2
       Vertex({ XMFLOAT3(+1.0f, -1.0f, -1.0f), XMFLOAT4(Colors::Green) }), // 3
       Vertex({ XMFLOAT3(-1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Blue) }),  // 4
       Vertex({ XMFLOAT3(-1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Yellow) }),// 5
       Vertex({ XMFLOAT3(+1.0f, +1.0f, +1.0f), XMFLOAT4(Colors::Cyan) }),  // 6
       Vertex({ XMFLOAT3(+1.0f, -1.0f, +1.0f), XMFLOAT4(Colors::Magenta) })// 7
    };
    std::array<std::uint16_t, 36> indices = {
       0, 1, 2,                    0, 2, 3, // front face
       4, 6, 5,                    4, 7, 6, // back face
       4, 5, 1,                    4, 1, 0, // left face
       3, 2, 6,                    3, 6, 7, // right face
       1, 5, 6,                    1, 6, 2, // top face
       4, 0, 3,                    4, 3, 7// bottom face
    };
```





BoxApp Class - Definition (12)

```
// BoxApp.cpp
    const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
    const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

    mBoxGeo = std::make_unique<MeshGeometry>();
    mBoxGeo->Name = "boxGeo";

    ThrowIfFailed(D3DCreateBlob(vbByteSize, &mBoxGeo->VertexBufferCPU));
    CopyMemory(mBoxGeo->VertexBufferCPU->GetBufferPointer(),
        vertices.data(), vbByteSize);

    ThrowIfFailed(D3DCreateBlob(ibByteSize, &mBoxGeo->IndexBufferCPU));
    CopyMemory(mBoxGeo->IndexBufferCPU->GetBufferPointer(),
        indices.data(), ibByteSize);

    mBoxGeo->VertexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
        mCommandList.Get(), vertices.data(), vbByteSize,
        mBoxGeo->VertexBufferUploader);

    mBoxGeo->IndexBufferGPU = d3dUtil::CreateDefaultBuffer(md3dDevice.Get(),
        mCommandList.Get(), indices.data(), ibByteSize,
        mBoxGeo->IndexBufferUploader);
```

BoxApp Class - Definition (13)

```
// BoxApp.cpp
    mBoxGeo->VertexByteStride = sizeof(Vertex);
    mBoxGeo->VertexBufferByteSize = vbByteSize;
    mBoxGeo->IndexFormat = DXGI_FORMAT_R16_UINT;
    mBoxGeo->IndexBufferByteSize = ibByteSize;

    SubmeshGeometry submesh;
    submesh.IndexCount = (UINT)indices.size();
    submesh.StartIndexLocation = 0;
    submesh.BaseVertexLocation = 0;

    mBoxGeo->DrawArgs["box"] = submesh;
}
```

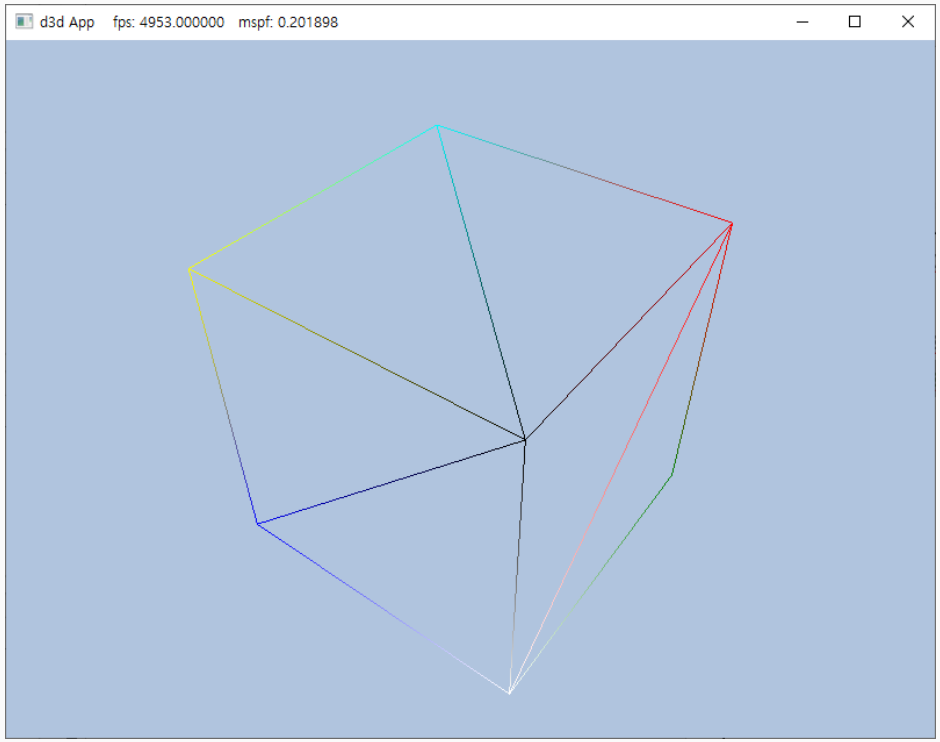
BoxApp Class - Definition (14)

```
// BoxApp.cpp
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)));
}
```

Wire Frame

```
void BoxApp::BuildPSO(){
// …
    psoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
    psoDesc.RasterizerState.FillMode = D3D12_FILL_MODE_WIREFRAME;
// …
}
```



Rotating World Matrix (1)

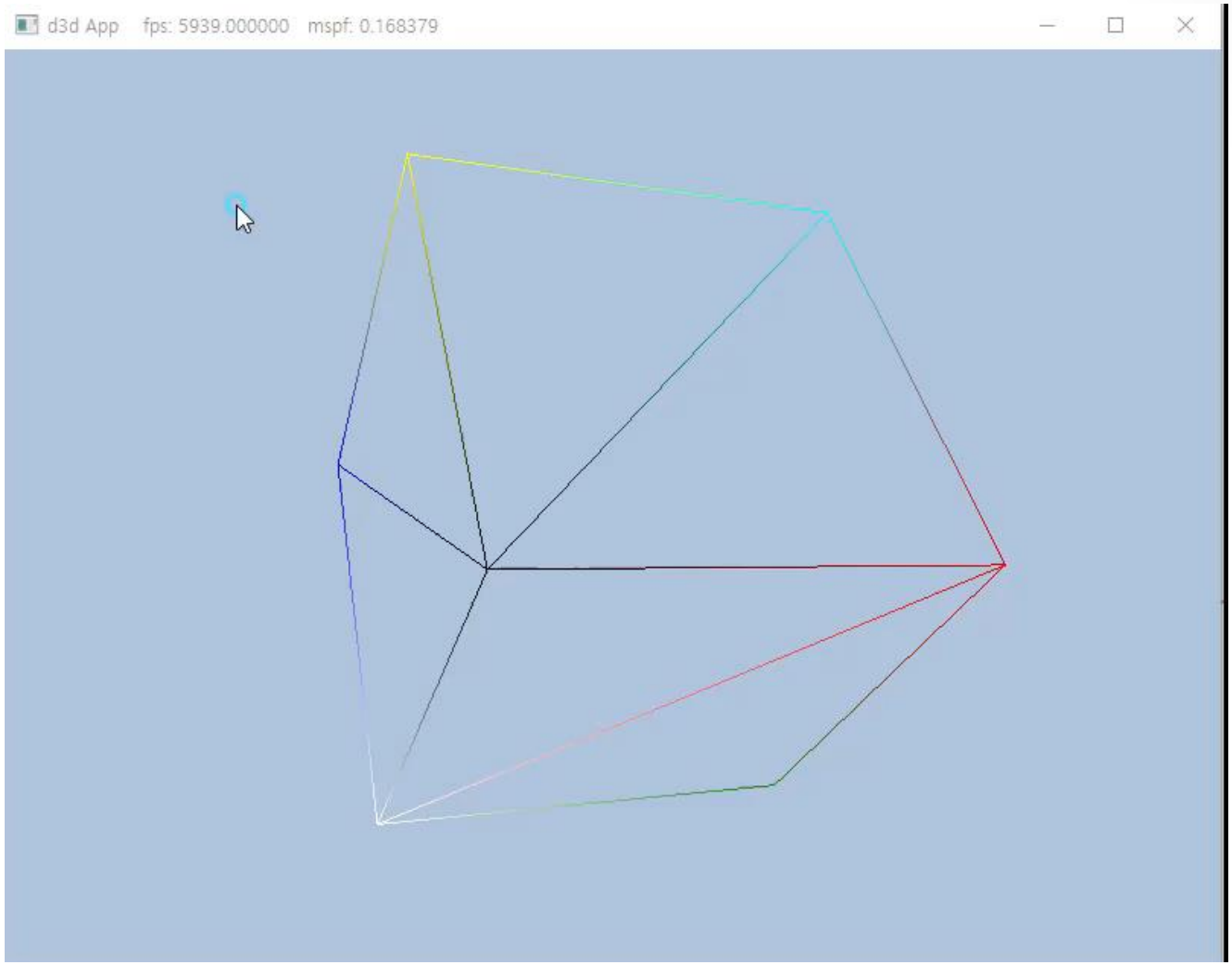
```
void BoxApp::Update(const GameTimer& gt) {
// …
    XMMATRIX view = XMMatrixLookAtLH(pos, target, up);
    XMStoreFloat4x4(&mView, view);

    XMMATRIX world = XMLoadFloat4x4(&mWorld);

    float angle = static_cast<float>(gt.TotalTime() * 90.0);
    const XMVECTOR rotationAxis
        = XMVector3Normalize(XMVectorSet(0.2, 0.3, 0.5, 0));
    world = XMMatrixRotationAxis(rotationAxis, XMConvertToRadians(angle));

    XMMATRIX proj = XMLoadFloat4x4(&mProj);
    XMMATRIX worldViewProj = world*view*proj;
// …
}
```

Rotating World Matrix (2)

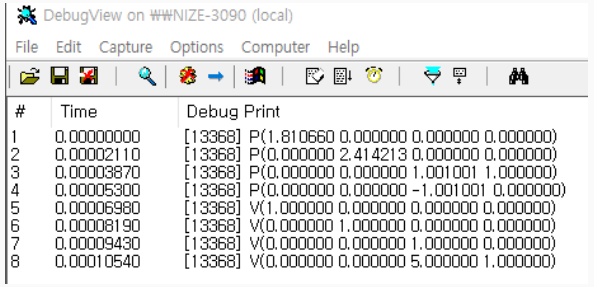


Formatted String for `OutputDebugString` (1)

```
// d2dUtil.h
class d3dUtil {
public:
    // insertion
     static int VDebugPrintF(const char* format, va_list argList) {
         const unsigned int MAX_CHARS = 1024;
         static char s_buffer[MAX_CHARS];
         int charsWritten
             = vsnprintf(s_buffer, MAX_CHARS, format, argList);
         OutputDebugStringA(s_buffer);
         return charsWritten;
     }
     static int DebugPrintF(const char* format, ...) {
         va_list argList;
         va_start(argList, format);
         int charsWritten = VDebugPrintF(format, argList);
         va_end(argList);
         return charsWritten;
     }
// …
};
```

Formatted String for `OutputDebugString` (2)

```
// BoxApp.cpp
void BoxApp::OnResize() {
// …
    for(int i = 0 ; i < 4 ; ++i)
        d3dUtil::DebugPrintF("P(%lf %lf %lf %lf)", P.r[i].m128_f32[0],
            P.r[i].m128_f32[1], P.r[i].m128_f32[2], P.r[i].m128_f32[3]);
    float x = 0.f;    float z = -5.f;    float y = 0.f;
    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);
    for (int i = 0; i < 4; ++i)
        d3dUtil::DebugPrintF("V(%lf %lf %lf %lf)", view.r[i].m128_f32[0],
            view.r[i].m128_f32[1], view.r[i].m128_f32[2],
            view.r[i].m128_f32[3]);
}
```



Kyung Hee University
nize@khu.ac.kr