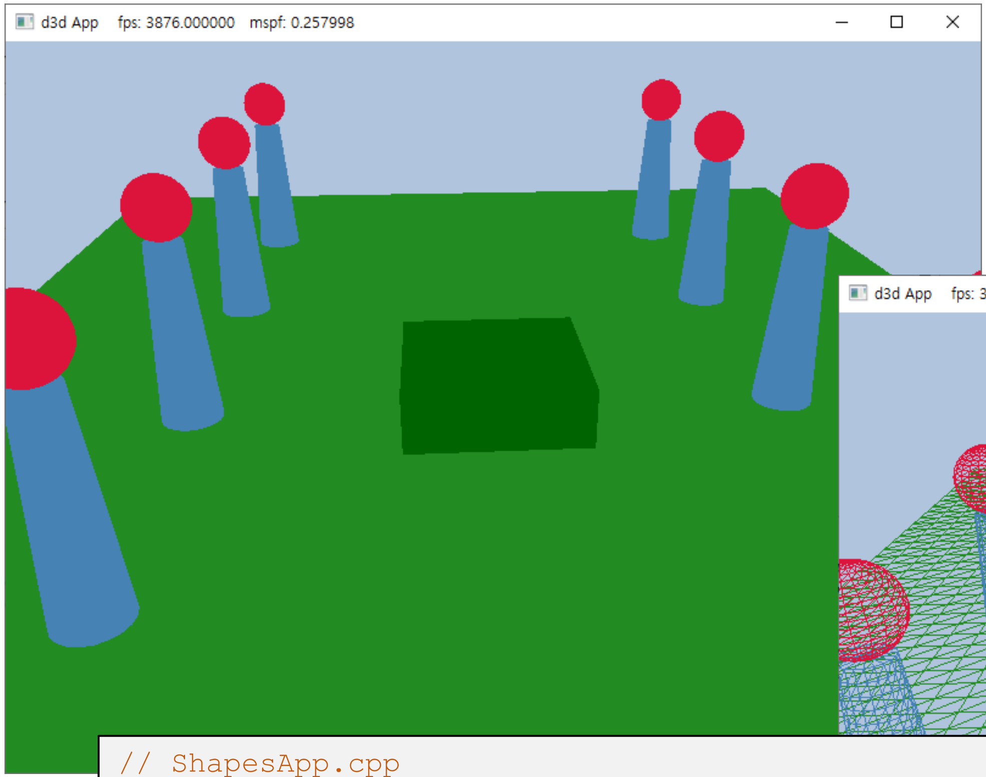
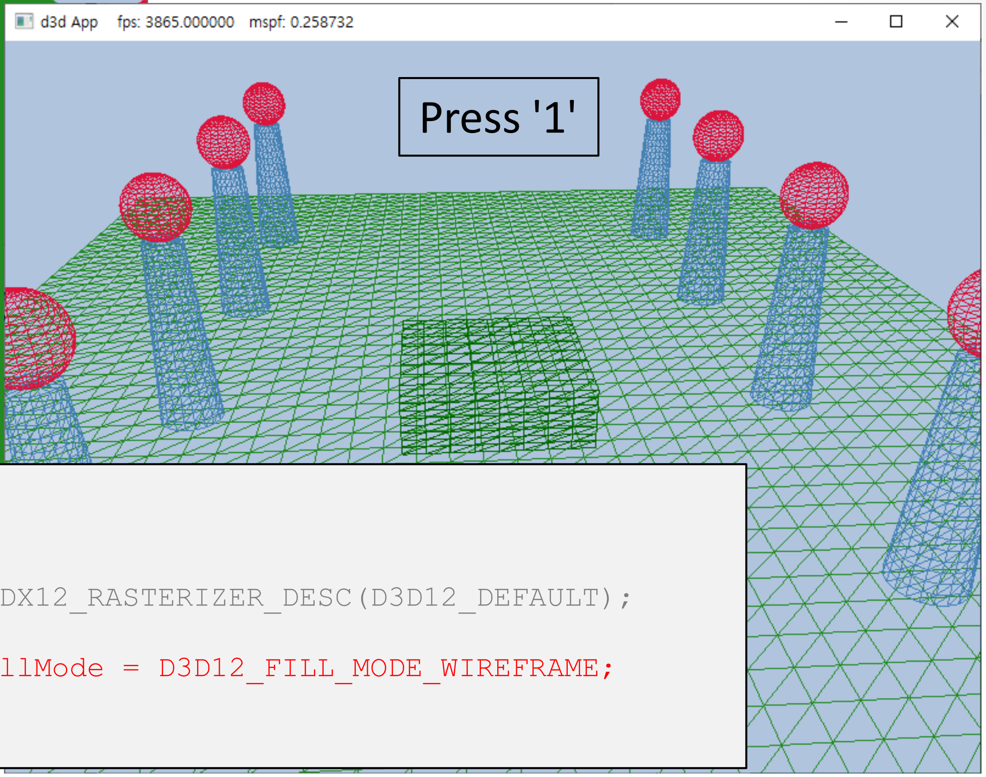
II. Direct3D Foundations
7. Drawing in Direct3D Part II

Game Graphic Programming
Kyung Hee University
Software Convergence
Prof. Daeho Lee

Drawing Common Shapes





```
// ShapesApp.cpp
void ShapesApp::BuildPSOs() {
// …
      opaquePsoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
      // deletion
      // opaquePsoDesc.RasterizerState.FillMode = D3D12_FILL_MODE_WIREFRAME;
// …
};
```

Frame Resources (1)

- The CPU and GPU work in parallel.
  - So far in our demos, we have been synchronizing the CPU and GPU once per frame.
  - Thus 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.
  - This solution works but is inefficient for the following reasons:
    - 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.
  - One solution to this problem is to create a circular array of the resources the CPU needs to modify each frame. We call such resources frame resources, and we usually use a circular array of three frame resource elements.

Frame Resources (2)

```
// Box Demo
void BoxApp::Draw(const GameTimer& gt) {
// …
    FlushCommandQueue();
}

// After Chapter 7
// $(ProjectDir)FrameResource.h (cpp)
// struct FrameResource 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();
```

Frame Resources (3)

```
    // 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;
};

// Application class
std::vector<std::unique_ptr<FrameResource>> mFrameResources; // member
void BuildFrameResources(); // member

const int gNumFrameResources = 3; // global variable
```

Frame Resources (4)

```
// Application class
void ShapesApp::Update(const GameTimer& gt) {
    OnKeyboardInput(gt);
    UpdateCamera(gt);

    // Cycle through the circular frame resource array.
    mCurrFrameResourceIndex = (mCurrFrameResourceIndex + 1) % gNumFrameResources;
    mCurrFrameResource = mFrameResources[mCurrFrameResourceIndex].get();

    // Has the GPU finished processing the commands of the current frame resource?
    // If not, wait until the GPU has completed commands up to this fence point.
    if(mCurrFrameResource->Fence != 0
      && mFence->GetCompletedValue() < mCurrFrameResource->Fence) {
        HANDLE eventHandle = CreateEventEx(nullptr, false, false, EVENT_ALL_ACCESS);
        ThrowIfFailed
          (mFence->SetEventOnCompletion(mCurrFrameResource->Fence, eventHandle));
        WaitForSingleObject(eventHandle, INFINITE);
        CloseHandle(eventHandle);
    }

    UpdateObjectCBs(gt);
    UpdateMainPassCB(gt);
}
```

Render Items (1)

- Drawing an object requires setting multiple parameters such as binding vertex and index buffers, binding object constants, setting a primitive type.
- **`struct RenderItem`**: A lightweight structure that stores the data needed to draw an object.

Render Items (2)

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

    XMFLOAT4X4 World = MathHelper::Identity4x4();

    // Dirty flag indicating the object data has changed, and we need
    // to update the constant buffer.
    // NumFramesDirty = gNumFrameResources so that each frame resource gets the update.
    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;
};
```

The dirty flag pattern is a design pattern that tracks changes to an object's state by maintaining a flag indicating whenever the object's properties have been modified.

Pass Constants (1)

```
// HLSL
cbuffer cbPerObject : register(b0) { // CB per object (world matrix)
   float4x4 gWorld;
};

cbuffer cbPass : register(b1) { // CB for view, projection, time …
    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;
};
```

Constant buffer padding occurs so that elements are packed into 4D vectors, with the restriction that a single element cannot be split across two 4D vectors.

Pass Constants (2)

```
// C++
struct ObjectConstants { // CB per object (world matrix)
    DirectX::XMFLOAT4X4 World = MathHelper::Identity4x4();
};
struct PassConstants { // CB for view, projection, time …
    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;
};
```

Pass Constants (3)

- **`FrameResource`** structure

```
struct FrameResource {
// …
// 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;
// …
};
FrameResource::FrameResource(ID3D12Device* device, UINT passCount,
   UINT objectCount) {
// …
     PassCB = std::make_unique<UploadBuffer<PassConstants>>(device,
       passCount, true); // passCount ← 1
     ObjectCB = std::make_unique<UploadBuffer<ObjectConstants>>(device,
       objectCount, true);
   // objectCount ← # of objects, (UINT)mAllRitems.size()
}
```

```
UploadBuffer(ID3D12Device* device, UINT elementCount, bool isConstantBuffer);
```

Application & Updating

```
class ShapesApp : public D3DApp{
// …
   std::vector<std::unique_ptr<FrameResource>> mFrameResources;
   FrameResource* mCurrFrameResource = nullptr;
   int mCurrFrameResourceIndex = 0;
// …
};

void ShapesApp::Update(const GameTimer& gt) {
// …
   mCurrFrameResourceIndex
     = (mCurrFrameResourceIndex + 1) % gNumFrameResources;
   mCurrFrameResource = mFrameResources[mCurrFrameResourceIndex].get();

   UpdateObjectCBs(gt);
   UpdateMainPassCB(gt);
}
```

Updating Object CBs

```
void ShapesApp::UpdateObjectCBs(const GameTimer& gt) {
// …
    auto currObjectCB = mCurrFrameResource->ObjectCB.get();
    for(auto& e : mAllRitems) {
     if(e->NumFramesDirty > 0) {
        XMMATRIX world = XMLoadFloat4x4(&e->World);
        ObjectConstants objConstants;
        XMStoreFloat4x4(&objConstants.World, XMMatrixTranspose(world));

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

        e->NumFramesDirty--;
      }
   }
}
```

```
void UploadBuffer::CopyData(int elementIndex, const T& data){
    memcpy(&mMappedData[elementIndex*mElementByteSize], &data, sizeof(T));
}
```

Updating Pass Constants

```
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));

// …

    auto currPassCB = mCurrFrameResource->PassCB.get();
    currPassCB->CopyData(0, mMainPassCB);
}
```

Root Signature

```
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];

     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);

    // …
}
```

Kyung Hee University
nize@khu.ac.kr

Geometry Generator (1)

- Creating the geometry for ellipsoids, spheres, cylinders, and cones.
- /Common/GeometryGenerator.cpp/h

```
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; // for normal mapping
        DirectX::XMFLOAT2 TexC;     // u, v
    };
```

Geometry Generator (3)

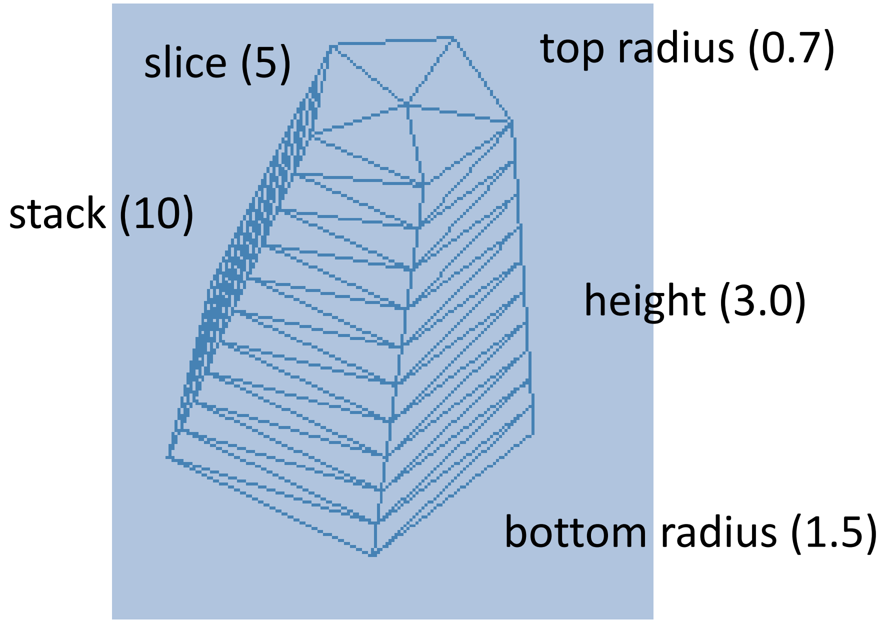
```
    struct MeshData {
       std::vector<Vertex> Vertices;
       std::vector<uint32> Indices32;

       std::vector<uint16>& GetIndices16() {
          // …
          return mIndices16;
       }
    private:
        std::vector<uint16> mIndices16;
    };
    MeshData CreateBox(float width, float height,
       float depth, uint32 numSubdivisions);
    MeshData CreateSphere(float radius, uint32 sliceCount,
       uint32 stackCount);
    MeshData CreateGeosphere(float radius, uint32 numSubdivisions);
    MeshData CreateCylinder(float bottomRadius, float topRadius,
        float height, uint32 sliceCount, uint32 stackCount);
    MeshData CreateGrid(float width, float depth, uint32 m, uint32 n);
    MeshData CreateQuad(float x, float y, float w, float h,
        float depth);
    // …
};
```

•

Generating a Cylinder Mesh

- Cylinder: bottom/top radii, height, slice and stack.

```
GeometryGenerator geoGen;
GeometryGenerator::MeshData cylinder
   = geoGen.CreateCylinder(1.5f, 0.7f, 3.0f, 5, 10);
```

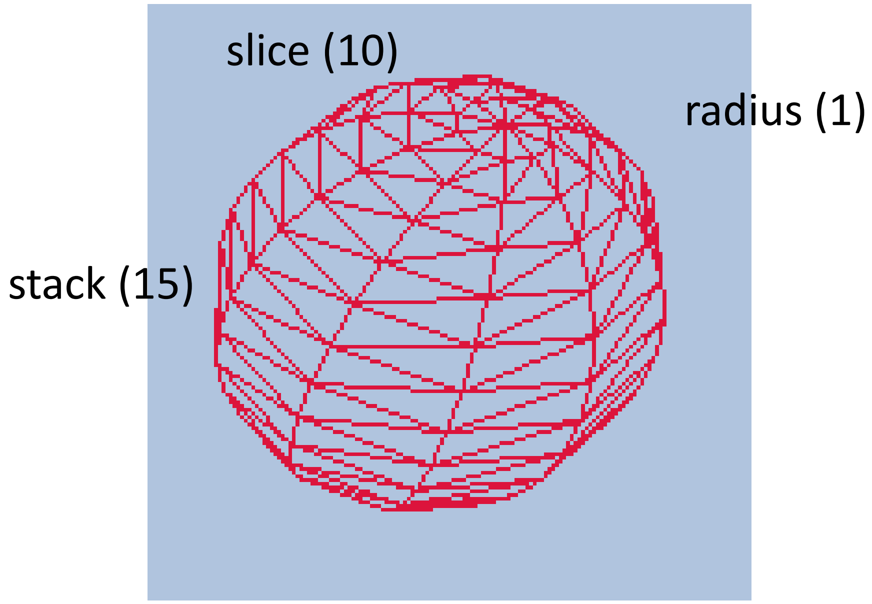


Kyung Hee University
nize@khu.ac.kr

Generating a Sphere Mesh

- Sphere: radius, slice and stack.

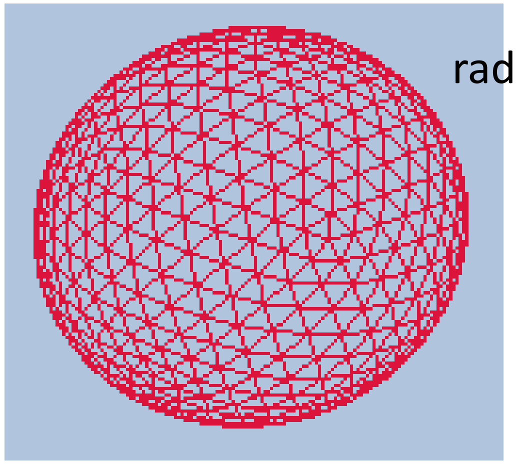
```
GeometryGenerator geoGen;
GeometryGenerator::MeshData sphere
   = geoGen.CreateSphere(1.0f, 10, 15);
```



Generating a Geosphere Mesh

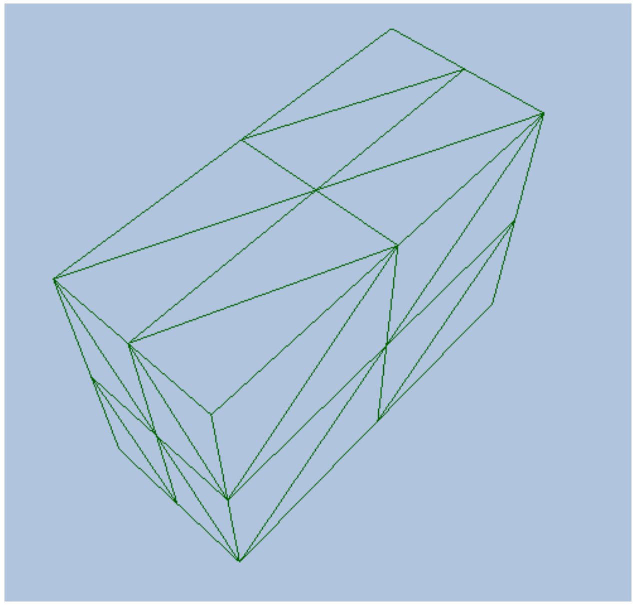
- A geosphere approximates a sphere using triangles with almost equal areas as well as equal side lengths.
  - # of subdivisions: subdividing each triangle into four triangles each iteration (tessellation, LOD (level-of-detail)).

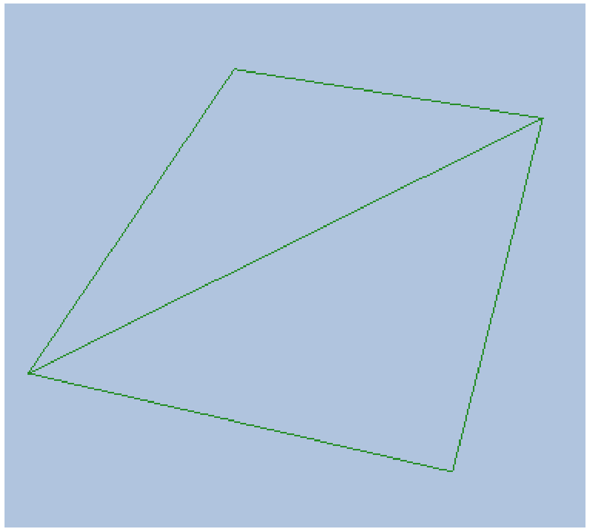
```
GeometryGenerator geoGen;
GeometryGenerator::MeshData sphere
   = geoGen.CreateGeosphere(1.0f, 3);
```



Generating Box & Grid

```
MeshData CreateBox(float width, float height, float depth,
   uint32 numSubdivisions);
MeshData CreateGrid(float width, float depth, uint32 m, uint32 n);
        // n & m: # of vertices
geoGen.CreateBox(3.f, 5.f, 7.f, 1); geoGen.CreateGrid(5.f, 7.f, 2, 2);
```





Vertex & Index Buffers (1)

\Chapter 7 Drawing in Direct3D Part II\Shapes

```
class ShapesApp : public D3DApp {
// …
};

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();
```

Vertex & Index Buffers (2)

```
 // 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();

   // 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;
```

Vertex & Index Buffers (3)

```
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;
```

Vertex & Index Buffers (4)

```
// Extract the vertex elements we are interested in and pack the
// vertices of all the meshes into one vertex buffer.
auto totalVertexCount = box.Vertices.size() + grid.Vertices.size() +
    sphere.Vertices.size() + cylinder.Vertices.size();

std::vector<Vertex> vertices(totalVertexCount);
UINT k = 0;
for(size_t i = 0; i < box.Vertices.size(); ++i, ++k) {
    vertices[k].Pos = box.Vertices[i].Position;
    vertices[k].Color = XMFLOAT4(DirectX::Colors::DarkGreen);
}
for(size_t i = 0; i < grid.Vertices.size(); ++i, ++k) {
    vertices[k].Pos = grid.Vertices[i].Position;
    vertices[k].Color = XMFLOAT4(DirectX::Colors::ForestGreen);
}
for(size_t i = 0; i < sphere.Vertices.size(); ++i, ++k) {
    vertices[k].Pos = sphere.Vertices[i].Position;
    vertices[k].Color = XMFLOAT4(DirectX::Colors::Crimson);
}
for(size_t i = 0; i < cylinder.Vertices.size(); ++i, ++k) {
    vertices[k].Pos = cylinder.Vertices[i].Position;
    vertices[k].Color = XMFLOAT4(DirectX::Colors::SteelBlue);
}
```

```
struct VertexIn{
 float3 PosL : POSITION;
 float4 Color: COLOR;
}; // hlsl
```

Vertex & Index Buffers (5)

```
std::vector<std::uint16_t> indices;
indices.insert(indices.end(),
    std::begin(box.GetIndices16()), std::end(box.GetIndices16()));
indices.insert(indices.end(),
    std::begin(grid.GetIndices16()), std::end(grid.GetIndices16()));
indices.insert(indices.end(),
    std::begin(sphere.GetIndices16()), std::end(sphere.GetIndices16()));
indices.insert(indices.end(),
    std::begin(cylinder.GetIndices16()), std::end(cylinder.GetIndices16()));

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

auto geo = std::make_unique<MeshGeometry>();
geo->Name = "shapeGeo";
ThrowIfFailed(D3DCreateBlob(vbByteSize, &geo->VertexBufferCPU));
CopyMemory(geo->VertexBufferCPU->GetBufferPointer(),
    vertices.data(), vbByteSize);

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

Vertex & Index Buffers (6)

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

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

    geo->VertexByteStride = sizeof(Vertex);
    geo->VertexBufferByteSize = vbByteSize;
    geo->IndexFormat = DXGI_FORMAT_R16_UINT;
    geo->IndexBufferByteSize = ibByteSize;

    geo->DrawArgs["box"] = boxSubmesh;
    geo->DrawArgs["grid"] = gridSubmesh;
    geo->DrawArgs["sphere"] = sphereSubmesh;
    geo->DrawArgs["cylinder"] = cylinderSubmesh;

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

Render Items (1)

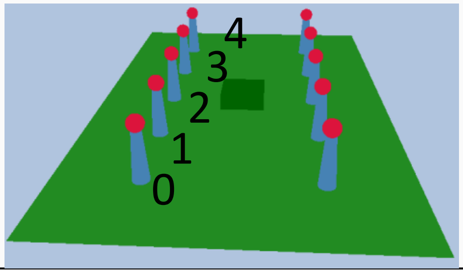
```
struct RenderItem{ /* … */ };
class ShapesApp : public D3DApp { // …
    // List of all the render items.
    std::vector<std::unique_ptr<RenderItem>> mAllRitems;
    // Render items divided by PSO.
    std::vector<RenderItem*> mOpaqueRitems;
// …
};
void ShapesApp::BuildRenderItems() {
    auto 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));
```

Render Items (2)

```
auto gridRitem = std::make_unique<RenderItem>();
gridRitem->World = MathHelper::Identity4x4();
gridRitem->ObjCBIndex = 1;
gridRitem->Geo = mGeometries["shapeGeo"].get();
gridRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
gridRitem->IndexCount = gridRitem->Geo->DrawArgs["grid"].IndexCount;

gridRitem->StartIndexLocation
   = gridRitem->Geo->DrawArgs["grid"].StartIndexLocation;
gridRitem->BaseVertexLocation
   = gridRitem->Geo->DrawArgs["grid"].BaseVertexLocation;

mAllRitems.push_back(std::move(gridRitem));
```

Render Items (3)

```
UINT objCBIndex = 2;
for(int i = 0; i < 5; ++i) {
    auto leftCylRitem = std::make_unique<RenderItem>();
    auto rightCylRitem = std::make_unique<RenderItem>();
    auto leftSphereRitem = std::make_unique<RenderItem>();
    auto rightSphereRitem = std::make_unique<RenderItem>();

    XMMATRIX leftCylWorld
        = XMMatrixTranslation(-5.0f, 1.5f, -10.0f + i*5.0f);
    XMMATRIX rightCylWorld
        = XMMatrixTranslation(+5.0f, 1.5f, -10.0f + i*5.0f);
    XMMATRIX leftSphereWorld
        = XMMatrixTranslation(-5.0f, 3.5f, -10.0f + i*5.0f);
    XMMATRIX rightSphereWorld
        = XMMatrixTranslation(+5.0f, 3.5f, -10.0f + i*5.0f);
```



Render Items (4)

```
XMStoreFloat4x4(&leftCylRitem->World, rightCylWorld);
leftCylRitem->ObjCBIndex = objCBIndex++;
leftCylRitem->Geo = mGeometries["shapeGeo"].get();
leftCylRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
leftCylRitem->IndexCount
   = leftCylRitem->Geo->DrawArgs["cylinder"].IndexCount;
leftCylRitem->StartIndexLocation
   = leftCylRitem->Geo->DrawArgs["cylinder"].StartIndexLocation;
leftCylRitem->BaseVertexLocation
   = leftCylRitem->Geo->DrawArgs["cylinder"].BaseVertexLocation;

XMStoreFloat4x4(&rightCylRitem->World, leftCylWorld);
rightCylRitem->ObjCBIndex = objCBIndex++;
rightCylRitem->Geo = mGeometries["shapeGeo"].get();
rightCylRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
rightCylRitem->IndexCount
   = rightCylRitem->Geo->DrawArgs["cylinder"].IndexCount;
rightCylRitem->StartIndexLocation
   = rightCylRitem->Geo->DrawArgs["cylinder"].StartIndexLocation;
rightCylRitem->BaseVertexLocation
   = rightCylRitem->Geo->DrawArgs["cylinder"].BaseVertexLocation;
```

Render Items (5)

```
        XMStoreFloat4x4(&leftSphereRitem->World, leftSphereWorld);
        leftSphereRitem->ObjCBIndex = objCBIndex++;
        leftSphereRitem->Geo = mGeometries["shapeGeo"].get();
        leftSphereRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
        leftSphereRitem->IndexCount
           = leftSphereRitem->Geo->DrawArgs["sphere"].IndexCount;
        leftSphereRitem->StartIndexLocation
           = leftSphereRitem->Geo->DrawArgs["sphere"].StartIndexLocation;
        leftSphereRitem->BaseVertexLocation
           = leftSphereRitem->Geo->DrawArgs["sphere"].BaseVertexLocation;

        XMStoreFloat4x4(&rightSphereRitem->World, rightSphereWorld);
        rightSphereRitem->ObjCBIndex = objCBIndex++;
        rightSphereRitem->Geo = mGeometries["shapeGeo"].get();
        rightSphereRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST;
        rightSphereRitem->IndexCount
           = rightSphereRitem->Geo->DrawArgs["sphere"].IndexCount;
        rightSphereRitem->StartIndexLocation
           = rightSphereRitem->Geo->DrawArgs["sphere"].StartIndexLocation;
        rightSphereRitem->BaseVertexLocation
           = rightSphereRitem->Geo->DrawArgs["sphere"].BaseVertexLocation;
```

Render Items (6)

```
        mAllRitems.push_back(std::move(leftCylRitem));
        mAllRitems.push_back(std::move(rightCylRitem));
        mAllRitems.push_back(std::move(leftSphereRitem));
        mAllRitems.push_back(std::move(rightSphereRitem));
    }

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

Frame Resources & Constant Buffer Views (1)

```
struct FrameResource {
// …
    std::unique_ptr<UploadBuffer<PassConstants>> PassCB = nullptr;
    std::unique_ptr<UploadBuffer<ObjectConstants>> ObjectCB = nullptr;

// …
};

class ShapesApp : public D3DApp {
// …
 std::vector<std::unique_ptr<FrameResource>> mFrameResources;
// …
};
```

Frame Resources & Constant Buffer Views (2)

```
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 3 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)));
}
```

Frame Resources & Constant Buffer Views (3)

```
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;
```

Frame Resources & Constant Buffer Views (4)

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

Frame Resources & Constant Buffer Views (5)

```
    UINT passCBByteSize
       = d3dUtil::CalcConstantBufferByteSize(sizeof(PassConstants));

    // Last three descriptors are the pass CBVs for each frame resource.
    for(int frameIndex = 0; frameIndex < gNumFrameResources; ++frameIndex) {
        auto passCB = mFrameResources[frameIndex]->PassCB->Resource();
        D3D12_GPU_VIRTUAL_ADDRESS cbAddress = passCB->GetGPUVirtualAddress();

        // Offset to the pass cbv in the descriptor heap.
        int heapIndex = mPassCbvOffset + frameIndex;
        auto handle
= CD3DX12_CPU_DESCRIPTOR_HANDLE(mCbvHeap->GetCPUDescriptorHandleForHeapStart());
        handle.Offset(heapIndex, mCbvSrvUavDescriptorSize);

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

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

Drawing the Scenes (1)

```
void ShapesApp::Draw(const GameTimer& gt) {
// …
    DrawRenderItems(mCommandList.Get(), mOpaqueRitems);
// …
}

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

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

Drawing the Scenes (2)

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

Toggle for Wire Frame & Solid

```
void ShapesApp::OnKeyboardInput(const GameTimer& gt) {
    if(GetAsyncKeyState('1') & 0x8000)
        mIsWireframe = true;
    else
        mIsWireframe = false;
}

void ShapesApp::BuildPSOs() {
// …
   opaquePsoDesc.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
   // deletion
   // opaquePsoDesc.RasterizerState.FillMode = D3D12_FILL_MODE_WIREFRAME;
};
```

Root Parameters (1)

- Root parameter types
  - Descriptor table
    - It expects a descriptors table referencing a contiguous range in a heap that identifies the resource to be bound.
    - 1 DWORD (unsigned long)
  - Root descriptor (inline descriptor)
    - It expects a descriptor to be set directly that 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.
    - 2 DWORDs (64-bit)
  - Root constant
    - It expects a list of 32-bit constant values to be bound directly.
    - 1 DWORD per 32-bit constant

The maximum size of a root signature is 64 DWORDs.

Root Parameters (2)

```
// D3D12_ROOT_PARAMETER

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;
```

In Chap. 6
We used
`CD3DX12_ROOT_PARAMETER::InitAsDescriptorTable(…)`.

Root Parameters (3)

```
// D3D12_ROOT_PARAMETER_TYPE D3D12_ROOT_PARAMETER::ParameterType
// D3D12_ROOT_PARAMETER_TYPE

typedef enum D3D12_ROOT_PARAMETER_TYPE {
  D3D12_ROOT_PARAMETER_TYPE_DESCRIPTOR_TABLE = 0,
  D3D12_ROOT_PARAMETER_TYPE_32BIT_CONSTANTS,
  D3D12_ROOT_PARAMETER_TYPE_CBV,
  D3D12_ROOT_PARAMETER_TYPE_SRV,
  D3D12_ROOT_PARAMETER_TYPE_UAV
};
```

Root Parameters (3)

```
// D3D12_SHADER_VISIBILITY D3D12_ROOT_PARAMETER::ShaderVisibility;
// D3D12_SHADER_VISIBILITY
// It specifies
// which shader programs this root parameter is visible to.
// Usually in this lecture, we specify D3D12_SHADER_VISIBILITY_ALL.

typedef enum D3D12_SHADER_VISIBILITY {
  D3D12_SHADER_VISIBILITY_ALL = 0,
  D3D12_SHADER_VISIBILITY_VERTEX = 1,
  D3D12_SHADER_VISIBILITY_HULL = 2,
  D3D12_SHADER_VISIBILITY_DOMAIN = 3,
  D3D12_SHADER_VISIBILITY_GEOMETRY = 4,
  D3D12_SHADER_VISIBILITY_PIXEL = 5,
  D3D12_SHADER_VISIBILITY_AMPLIFICATION = 6,
  D3D12_SHADER_VISIBILITY_MESH = 7
};
```

In Chap. 6 **`CD3DX12_ROOT_PARAMETER::InitAsDescriptorTable(…, D3D12_SHADER_VISIBILITY visibility = D3D12_SHADER_VISIBILITY_ALL)`**

Descriptor Tables (1)

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

typedef struct D3D12_ROOT_DESCRIPTOR_TABLE {
  UINT                          NumDescriptorRanges;
  const D3D12_DESCRIPTOR_RANGE *pDescriptorRanges;
} D3D12_ROOT_DESCRIPTOR_TABLE;
```

Descriptor Tables (2)

```
// pDescriptorRanges is an array of D3D12_DESCRIPTOR_RANGEs
typedef struct D3D12_DESCRIPTOR_RANGE {
  D3D12_DESCRIPTOR_RANGE_TYPE RangeType;
  UINT                        NumDescriptors;
  UINT                        BaseShaderRegister;
  UINT                        RegisterSpace;
  UINT                        OffsetInDescriptorsFromTableStart;
} D3D12_DESCRIPTOR_RANGE;
```

In Chap. 6 `CD3DX12_DESCRIPTOR_RANGE::Init();`

Kyung Hee University
nize@khu.ac.kr

Descriptor Tables (3)

- 1. **`RangeType`**: It indicates the type of descriptor.

```
typedef enum D3D12_DESCRIPTOR_RANGE_TYPE {
  D3D12_DESCRIPTOR_RANGE_TYPE_SRV = 0,
  D3D12_DESCRIPTOR_RANGE_TYPE_UAV,
  D3D12_DESCRIPTOR_RANGE_TYPE_CBV,
  D3D12_DESCRIPTOR_RANGE_TYPE_SAMPLER
};
```

- 2. **`NumDescriptors`**: It is the number of descriptors in the range.
- 3. **`BaseShaderRegister`**: It binds to the base shader register arguments. If you set **`NumDescriptors`** to 3, **`BaseShaderRegister`** to 1, and the range type is CBV, then you will be binding to HLSL registers (**`b1, b2, b3`**).

Descriptor Tables (4)

- 4. **`RegisterSpace`**: It gives you another dimension to specify shader registers.
  For example, the following two registers seem to overlap register slot **`t0`**, but they are different registers because they live in different spaces:

```
Texture2D gDiffuseMap : register(t0, space0);
Texture2D gNormalMap : register(t0, space1);
```

- 5. **`OffsetInDescriptorsFromTableStart`**: The offset of this range of descriptors from the start of the table.

Descriptor Tables (5)

```
// CD3DX12_DESCRIPTOR_RANGE: A helper structure to enable easy initialization
//     of a D3D12_DESCRIPTOR_RANGE structure.
// Examples: 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 (1)

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

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

Root Descriptors (2)

- 1. **`ShaderRegister`**: It binds to the shader register the descriptor 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`**.

Root Descriptors (3)

```
// Unlike descriptor tables which require us to set a descriptor
// handle in a descriptor heap, to set a root descriptor,
// we simply bind the virtual address of the resource directly.

UINT objCBByteSize =
   d3dUtil::CalcConstantBufferByteSize(sizeof (ObjectConstants));
D3D12_GPU_VIRTUAL_ADDRESS objCBAddress
   = objectCB->GetGPUVirtualAddress();

// Offset to the constants for this object in the buffer.
objCBAddress += ri->ObjCBIndex*objCBByteSize;
cmdList->SetGraphicsRootConstantBufferView(
  0, // root parameter index
  objCBAddress);
```

Root Constants (1)

```
// A root constant root parameter is further defined by filling
// out the DescriptorTable member of D3D12_ROOT_PARAMETER.

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

- 1. **`ShaderRegister`**:
  See **`D3D12_ROOT_DESCRIPTOR::ShaderRegister`**.
- 2. **`RegisterSpace`**: See **`D3D12_DESCRIPTOR_RANGE::RegisterSpace`**.
- 3. **`Num32BitValues`**: The number of 32-bit constants this root parameter expects.

Root Constants (2)

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

Root Constants (3)

- **`ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstants`** method has the following prototype:

```
void
ID3D12GraphicsCommandList::SetGraphicsRoot32BitConstants(
UINT RootParameterIndex,
  UINT Num32BitValuesToSet,
  const void *pSrcData,
  UINT DestOffsetIn32BitValues);
```

  - 1. **`RootParameterIndex`**: Index of the root parameter we are setting.
  - 2. **`Num32BitValuesToSet`**: The number of 32-bit values to set.
  - 3. **`pSrcData`**: Pointer to an array of 32-bit values to set.
  - 4. **`DestOffsetIn32BitValues`**: Offset in 32-bit values in the constant buffer.

Root Constants (4)

```
// HLSL code.
cbuffer cbSettings : register(b0) {
   // We cannot have an array entry in a constant buffer that gets
   // mapped onto root constants, so list each element.
   int gBlurRadius;

   // Support up to 11 blur weights.
   float w0;
   float w1;
   float w2;
   float w3;
   float w4;
   float w5;
   float w6;
   float w7;
   float w8;
   float w9;
   float w10;
 };
```