Week 10

Building A First Person Camera and Dynamic Indexing

Hooman Salamat

Objectives

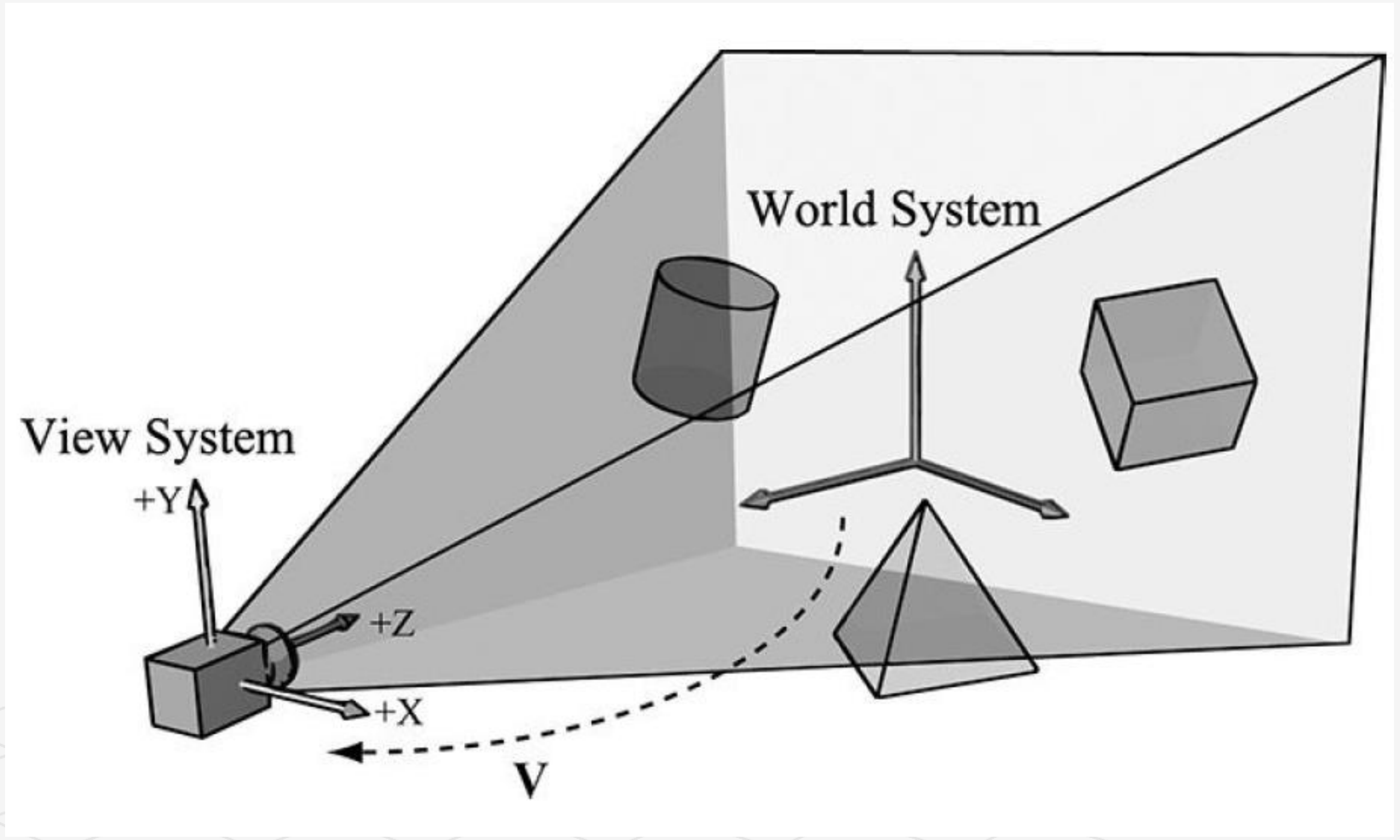
1. To review the mathematics of the view space transformation.
2. To be able to identify the typical functionality of a first person camera.
3. To learn how to implement a first person camera.
4. To understand how to dynamically index into an array of textures.

VIEW TRANSFORM REVIEW

View space is the coordinate system attached to the camera.

The camera sits at the origin looking down the positive *z*-axis, the *x*-axis aims to the right of the camera, and the *y*-axis aims above the camera.

The change of coordinate transformation from world space to view space is called the *view transform*, and the corresponding matrix is called the *view matrix*.



View Matrix

If $\mathbf{Q}_W = (Q_x, Q_y, Q_z, 1)$, $\mathbf{u}_W = (u_x, u_y, u_z, 0)$, $\mathbf{v}_W = (v_x, v_y, v_z, 0)$, and $\mathbf{w}_W = (w_x, w_y, w_z, 0)$ describe, respectively, the origin, *x*-, *y*-, and *z*-axes of view space with homogeneous coordinates relative to world space, the change of coordinate matrix from view space to world space is:

$$\mathbf{W} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

We want the reverse transformation from world space to view space $\mathbf{W}^{-1}$

The world matrix can be decomposed into a rotation followed by a translation:

$$\mathbf{V} = \mathbf{W}^{-1} = (\mathbf{RT})^{-1} = \mathbf{T}^{-1}\mathbf{R}^{-1} = \mathbf{T}^{-1}\mathbf{R}^{T}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -\mathbf{Q}\cdot\mathbf{u} & -\mathbf{Q}\cdot\mathbf{v} & -\mathbf{Q}\cdot\mathbf{w} & 1 \end{bmatrix}$$

THE CAMERA CLASS

The camera class stores:

1. The *position, right, up*

2. look vectors of the camera defining, respectively, the origin, *x*-axis, *y*-axis, and *z*-axis of the view space coordinate system in world coordinates, and the properties of the frustum.

The lens of the camera defines the frustum: its field of view and near and far planes

```
class Camera
{
public:

Camera();
~Camera();
private:
// Camera coordinate system with coordinates relative to world space.
DirectX::XMFLOAT3 mPosition = { 0.0f, 0.0f, 0.0f };
DirectX::XMFLOAT3 mRight = { 1.0f, 0.0f, 0.0f };
DirectX::XMFLOAT3 mUp = { 0.0f, 1.0f, 0.0f };
DirectX::XMFLOAT3 mLook = { 0.0f, 0.0f, 1.0f };

// Cache frustum properties.
float mNearZ = 0.0f;
float mFarZ = 0.0f;
float mAspect = 0.0f;
float mFovY = 0.0f;
float mNearWindowHeight = 0.0f;
float mFarWindowHeight = 0.0f;
```

Camera::GetPosition & Camera::SetLens methods

The client code does not need to convert if they need an XMVECTOR:

```
XMVECTOR Camera::GetPosition()const

{

return XMLoadFloat3(&mPosition);

}

XMFLOAT3 Camera::GetPosition3f()const

{

return mPosition;

}
```

We cache the frustum properties and build the projection matrix is the SetLens method:

```
void Camera::SetLens(float fovY, float aspect, float zn, float zf)
{
// cache properties
mFovY = fovY;
mAspect = aspect;
mNearZ = zn;
mFarZ = zf;

mNearWindowHeight = 2.0f * mNearZ * tanf( 0.5f*mFovY );
mFarWindowHeight  = 2.0f * mFarZ * tanf( 0.5f*mFovY );

XMMATRIX P = XMMatrixPerspectiveFovLH(mFovY, mAspect, mNearZ, mFarZ);
XMStoreFloat4x4(&mProj, P);
}
```

Derived Frustum Info

SetLens() caches the vertical field of view angle, but we additionally provide a method that derives the horizontal field of view angle.

Moreover, we provide methods to return the width and height of the frustum at the near and far planes.

```
float Camera::GetFovX()const
{
float halfWidth = 0.5f*GetNearWindowWidth();
return 2.0f*atan(halfWidth / mNearZ);
}

float Camera::GetNearWindowWidth()const
{
return mAspect * mNearWindowHeight;
}

float Camera::GetNearWindowHeight()const
{
return mNearWindowHeight;
}

float Camera::GetFarWindowWidth()const
{
return mAspect * mFarWindowHeight;
}

float Camera::GetFarWindowHeight()const
{
return mFarWindowHeight;
}
```

Transforming the Camera

For a first person camera, ignoring collision detection, we want to be able to:

1. Move the camera along its look vector to move forwards and backwards. This can be implemented by translating the camera position along its look vector.

2. Move the camera along its right vector to strafe right and left. This can be implemented by translating the camera position along its right vector.

3. Rotate the camera around its right vector to look up and down. This can be implemented by rotating the camera's look and up vectors around its right vector using the `XMMatrixRotationAxis function.`

4. Rotate the camera around the world's *y*-axis (assuming the *y*-axis corresponds to the world's "up" direction) vector to look right and left. This can be implemented by rotating all the basis vectors around the world's *y*-axis using the `XMMatrixRotationY function.`

Transforming the Camera

```cpp
void Camera::Walk(float d)
{
// mPosition += d*mLook
XMVECTOR s = XMVectorReplicate(d);
XMVECTOR l = XMLoadFloat3(&mLook);
XMVECTOR p = XMLoadFloat3(&mPosition);
XMStoreFloat3(&mPosition,
XMVectorMultiplyAdd(s, l, p));

mViewDirty = true;
}
void Camera::Strafe(float d)
{
// mPosition += d*mRight
XMVECTOR s = XMVectorReplicate(d);
XMVECTOR r = XMLoadFloat3(&mRight);
XMVECTOR p = XMLoadFloat3(&mPosition);
XMStoreFloat3(&mPosition,
XMVectorMultiplyAdd(s, r, p));

mViewDirty = true;
}
```

```cpp
void Camera::Pitch(float angle)
{
// Rotate up and look vector about the right vector.

XMMATRIX R = XMMatrixRotationAxis(XMLoadFloat3(&mRight), angle);

XMStoreFloat3(&mUp,   XMVector3TransformNormal(XMLoadFloat3(&mUp), R));
XMStoreFloat3(&mLook, XMVector3TransformNormal(XMLoadFloat3(&mLook), R));

mViewDirty = true;
}

void Camera::RotateY(float angle)
{
// Rotate the basis vectors about the world y-axis.

XMMATRIX R = XMMatrixRotationY(angle);

XMStoreFloat3(&mRight,   XMVector3TransformNormal(XMLoadFloat3(&mRight), R));
XMStoreFloat3(&mUp, XMVector3TransformNormal(XMLoadFloat3(&mUp), R));
XMStoreFloat3(&mLook, XMVector3TransformNormal(XMLoadFloat3(&mLook), R));

mViewDirty = true;
}
```

Building the View Matrix

The UpdateViewMatrix method *reorthonormalizes* the camera’s right, up, and look vectors: it makes sure they are mutually orthogonal to each other and unit length.

The second part of this method computes the view transformation matrix.

```
void Camera::UpdateViewMatrix()
{
if(mViewDirty)
{
XMVECTOR R = XMLoadFloat3(&mRight);
XMVECTOR U = XMLoadFloat3(&mUp);
XMVECTOR L = XMLoadFloat3(&mLook);
XMVECTOR P =
XMLoadFloat3(&mPosition);
```

```
// Keep camera's axes orthogonal to each other and of unit length.
L = XMVector3Normalize(L);
U = XMVector3Normalize(XMVector3Cross(L, R));

// U, L already ortho-normal, so no need to normalize cross product.
R = XMVector3Cross(U, L);

// Fill in the view matrix entries.
float x = -XMVectorGetX(XMVector3Dot(P, R));
float y = -XMVectorGetX(XMVector3Dot(P, U));
float z = -XMVectorGetX(XMVector3Dot(P, L));

XMStoreFloat3(&mRight, R);
XMStoreFloat3(&mUp, U);
XMStoreFloat3(&mLook, L);

mView(0, 0) = mRight.x;
mView(1, 0) = mRight.y;
mView(2, 0) = mRight.z;
mView(3, 0) = x;

mView(0, 1) = mUp.x;
mView(1, 1) = mUp.y;
mView(2, 1) = mUp.z;
mView(3, 1) = y;

mView(0, 2) = mLook.x;
mView(1, 2) = mLook.y;
mView(2, 2) = mLook.z;
mView(3, 2) = z;

mView(0, 3) = 0.0f;
mView(1, 3) = 0.0f;
mView(2, 3) = 0.0f;
mView(3, 3) = 1.0f;

mViewDirty = false;
}
}
```

CAMERA DEMO: CameraAndDynamicIndexingApp::OnResize

We can now remove all the old variables from our application class that were related to the orbital camera system such as mPhi, mTheta, mRadius, mView, and mProj. We will add a member variable:

Camera mCam;

When the window is resized, we know longer rebuild the projection matrix explicitly, and instead delegate the work to the Camera class with SetLens:

```
void CameraAndDynamicIndexingApp::OnResize()

{

    D3DApp::OnResize();


mCamera.SetLens(0.25f*MathHelper::Pi, AspectRatio(), 1.0f, 1000.0f);

}
```

Update Scene: **OnKeyboardInput & OnMouseMove methods**

we handle keyboard input to move the camera:

```
void CameraAndDynamicIndexingApp::OnKeyboardInput(const
GameTimer& gt)
{
const float dt = gt.DeltaTime();

if(GetAsyncKeyState('W') & 0x8000)
mCamera.Walk(10.0f*dt);

if(GetAsyncKeyState('S') & 0x8000)
mCamera.Walk(-10.0f*dt);

if(GetAsyncKeyState('A') & 0x8000)
mCamera.Strafe(-10.0f*dt);

if(GetAsyncKeyState('D') & 0x8000)
mCamera.Strafe(10.0f*dt);

mCamera.UpdateViewMatrix();
}
```

In the OnMouseMove method, we rotate the camera's look direction:

```
void CameraAndDynamicIndexingApp::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));

mCamera.Pitch(dy);
mCamera.RotateY(dx);
    }

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

The camera demo

Use the ‘W’, ‘S’, ‘A’, and ‘D’ keys to move forward, backward, strafe left, and strafe right, respectively.

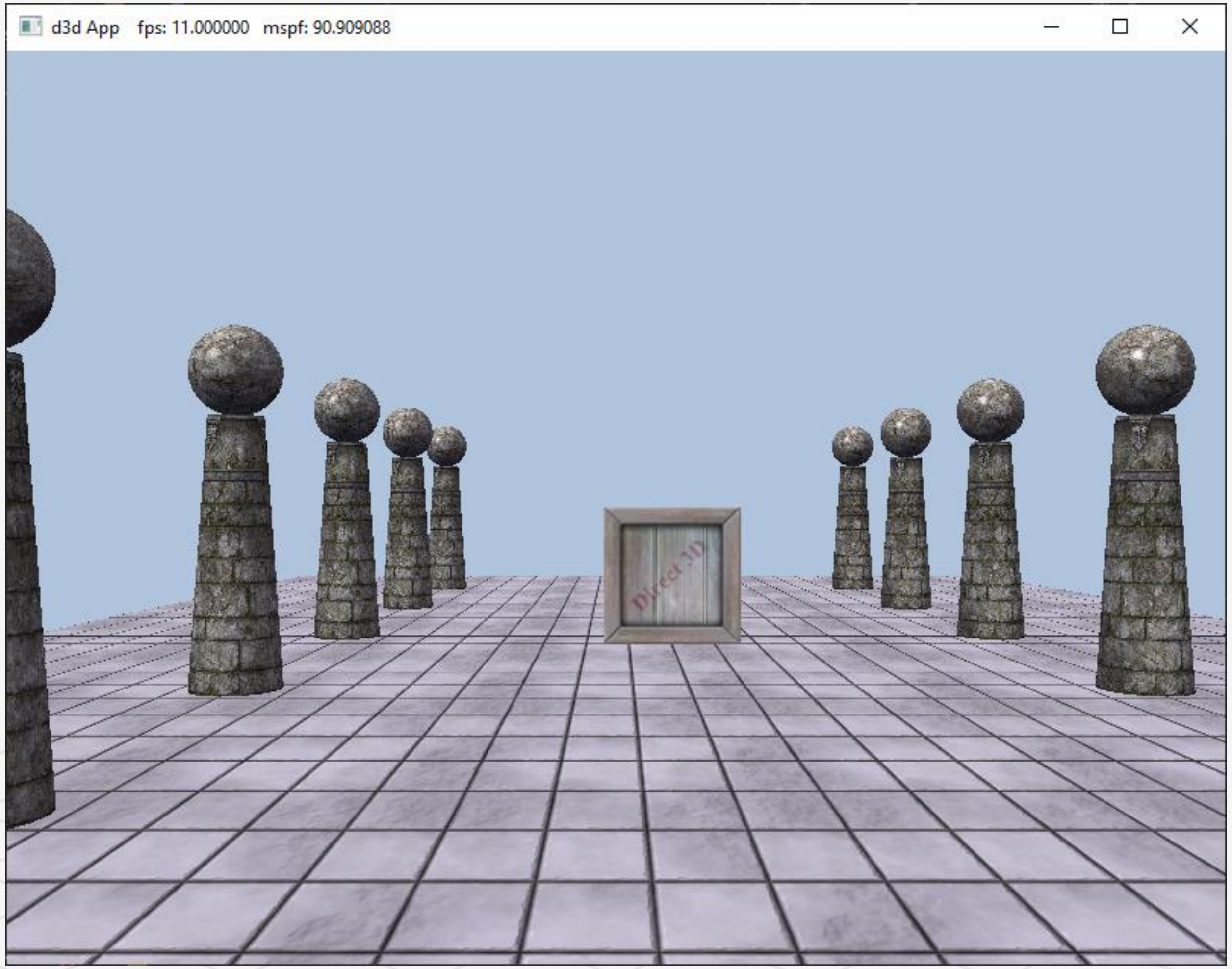
Hold the left mouse button down and move the mouse to “look” in different directions.

For rendering, the view and projection matrices can be accessed from the camera instance:

```
mCamera.UpdateViewMatrix();

XMMATRIX view = mCamera.View();

XMMATRIX proj = mCamera.Proj();
```



DYNAMIC INDEXING

We dynamically index into an array of resources in a shader program; in this demo, the resources will be an array of textures. The index can be specified in various ways:

1. The index can be an element in a constant buffer.

2. The index can be a system ID like SV_PrimitiveID, SV_VertexID, SV_DispatchThreadID, or SV_InstanceID.

3. The index can be the result of come calculation.

4. The index can come from a texture.

5. The index can come from a component of the vertex structure.

The following shader syntax declares a texture array of 4 elements and shows how we can index into the texture array where the index comes from a constant buffer:

```
cbuffer cbPerDrawIndex : register(b0)

{

int gDiffuseTexIndex;

};

Texture2D gDiffuseMap[4] : register(t0);

float4 texValue = gDiffuseMap[gDiffuseTexIndex].Sample( gsamLinearWrap, pin.TexC);
```

DYNAMIC INDEXING

We set the object constant buffer, the material constant buffer, and the diffuse texture map SRV on a per render-item basis.

How to minimize the number of descriptors we set on a per render-item basis? Why?

It will make our root signature smaller, which means less overhead per draw call;

With this setup, we only need to set a per object constant buffer for each render-item.

By dynamic indexing, we use the MaterialIndex to fetch the material to use for the draw call, and from that we use the DiffuseMapIndex to fetch the texture to use for the draw call.

1. Create a structured buffer that stores all of the material data. That is, instead of storing our material data in constant buffers, we will store it in a structured buffer. A structured buffer can be indexed in a shader program. This structured buffer will be bound to the rendering pipeline once per frame making all materials visible to the shader programs.

2. Add a MaterialIndex field to our object constant buffer to specify the index of the material to use for this draw call. In our shader programs, we use this to index into the material structured buffer.

3. Bind *all* of the texture SRV descriptors used in the scene once per frame, instead of binding one texture SRV per render-item.

4. Add a DiffuseMapIndex field to the material data that specifies the texture map associated with the material. We use this to index into the array of textures we bound to the pipeline in the previous step.

Dynamic Indexing

```
----FrameResource.h--------
struct MaterialData
{
DirectX::XMFLOAT4 DiffuseAlbedo = {
1.0f, 1.0f, 1.0f, 1.0f };
DirectX::XMFLOAT3 FresnelR0 = { 0.01f,
0.01f, 0.01f };
float Roughness = 64.0f;

// Used in texture mapping.
DirectX::XMFLOAT4X4 MatTransform =
MathHelper::Identity4x4();

UINT DiffuseMapIndex = 0;
UINT MaterialPad0;
UINT MaterialPad1;
UINT MaterialPad2;
};
```

We update the root signature based on the new data the shader expects as input:

```
void CameraAndDynamicIndexingApp::BuildRootSignature()
{
CD3DX12_DESCRIPTOR_RANGE texTable;
texTable.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 4, 0, 0);
CD3DX12_ROOT_PARAMETER slotRootParameter[4];
slotRootParameter[0].InitAsConstantBufferView(0);
    slotRootParameter[1].InitAsConstantBufferView(1);
    slotRootParameter[2].InitAsShaderResourceView(0, 1);
slotRootParameter[3].InitAsDescriptorTable(1, &texTable ,
D3D12_SHADER_VISIBILITY_PIXEL);
auto staticSamplers = GetStaticSamplers();

    // A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(4, slotRootParameter,
(UINT)staticSamplers.size(), staticSamplers.data(),
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

CameraAndDynamicIndexingApp::Draw

Now, before we draw any render-items, we can bind all of our materials and texture SRVs once per frame rather than per-render-item, and then each render-item just sets the object constant buffer:

```
void CameraAndDynamicIndexingApp::Draw(const GameTimer& gt)
{
auto passCB = mCurrFrameResource->PassCB->Resource();
mCommandList->SetGraphicsRootConstantBufferView(1, passCB-
>GetGPUVirtualAddress());
// Bind all the materials used in this scene.  For structured buffers,
we can bypass // the heap and set as a root descriptor.
auto matBuffer = mCurrFrameResource->MaterialBuffer->Resource();
mCommandList->SetGraphicsRootShaderResourceView(2, matBuffer-
>GetGPUVirtualAddress());
// Bind all the textures used in this scene.  Observe
    // that we only have to specify the first descriptor in the table.
    // The root signature knows how many descriptors are expected in
the table.
mCommandList->SetGraphicsRootDescriptorTable(3, mSrvDescriptorHeap-
>GetGPUDescriptorHandleForHeapStart());
DrawRenderItems(mCommandList.Get(), mOpaqueRitems);
```

```
void
CameraAndDynamicIndexingApp::DrawRenderItems(ID3D12GraphicsCommandLis
t* 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);

        D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB-
>GetGPUVirtualAddress() + ri->ObjCBIndex*objCBByteSize;

// CD3DX12_GPU_DESCRIPTOR_HANDLE tex(mSrvDescriptorHeap-
>GetGPUDescriptorHandleForHeapStart());
// tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);

cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);

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

The HLSL file

```
struct MaterialData
{
float4    DiffuseAlbedo;
float3    FresnelR0;
float     Roughness;
float4x4 MatTransform;
uint      DiffuseMapIndex;
uint      MatPad0;
uint      MatPad1;
uint      MatPad2;
};
```

```
// An array of textures, which is only supported in shader model 5.1+.  Unlike
Texture2DArray, the textures in this array can be different sizes and formats,
making it more flexible than texture arrays.

Texture2D gDiffuseMap[4] : register(t0);

// Put in space1, so the texture array does not overlap with these resources.

// The texture array will occupy registers t0, t1, ..., t3 in space0.

StructuredBuffer<MaterialData> gMaterialData : register(t0, space1);
```

The HLSL file

```
VertexOut VS(VertexIn vin)
{
VertexOut vout = (VertexOut)0.0f;
// Fetch the material data.
MaterialData matData = gMaterialData[gMaterialIndex];
    // Transform to world space.
    float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
    vout.PosW = posW.xyz;
    // Assumes nonuniform scaling; otherwise, need to use
inverse-transpose of world matrix.
    vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);
    // Transform to homogeneous clip space.
    vout.PosH = mul(posW, gViewProj);
// Output vertex attributes for interpolation across
triangle.
float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f),
gTexTransform);
vout.TexC = mul(texC, matData.MatTransform).xy;

    return vout;
}
```

```
float4 PS(VertexOut pin) : SV_Target

{

// Fetch the material data.

MaterialData matData = gMaterialData[gMaterialIndex];

float4 diffuseAlbedo = matData.DiffuseAlbedo;

float3 fresnelR0 = matData.FresnelR0;

float  roughness = matData.Roughness;

uint diffuseTexIndex = matData.DiffuseMapIndex;



// Dynamically look up the texture in the array.

diffuseAlbedo *= gDiffuseMap[diffuseTexIndex].Sample(gsamLinearWrap,
pin.TexC);
```

Additional uses of dynamic indexing

1. Merging nearby meshes with different textures into a single render-item so that they can be drawn with one draw call. The meshes could store the texture/material to use as an attribute in the vertex structure.

2. Multitexturing in a single rendering-pass where the textures have different sizes and formats.

3. Instancing render-items with different textures and materials using the `SV_InstanceID` value as an index. We will see an example of this in the next week.