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.

First-Person

In video games, first person is any graphical perspective rendered from the viewpoint of the player's character, or a viewpoint from the cockpit or front seat of a vehicle driven by the character.

Many genres incorporate first-person perspectives, among them adventure games, driving, sailing, and flight simulators.

 Most notable is the first-person shooter, in which the graphical perspective is an integral component of the gameplay.

Games with a first-person perspective are usually avatar-based, wherein the game displays what the player's avatar would see with the avatar's own eyes.

It is not clear exactly when the earliest such first-person shooter video game was created. There are two claimants, *Spasim* and *Maze War*.

Spasim is a 32-player (4 teams of 8 players) 3D networked space flight simulation game and first-person space shooter developed by Jim Bowery for the PLATO computer network and released in 1974.

Maze War is a 1973 computer game where players wander around a maze, being capable of moving backward or forwards, turning right or left in 90-degree increments, and peeking around corners through doorways. Steve Colley and Greg Thompson developed the MazeWars program at NASA.

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*.
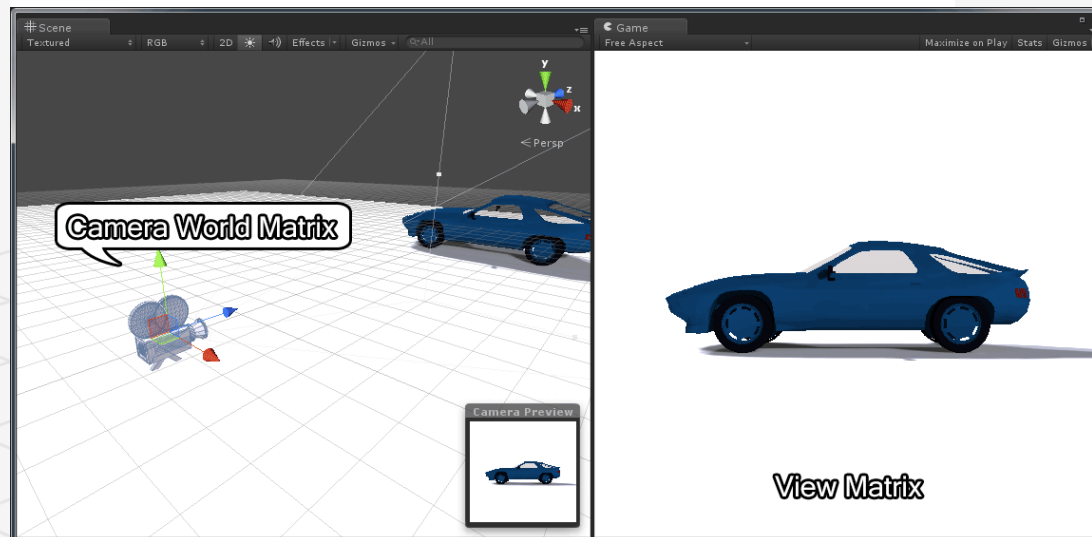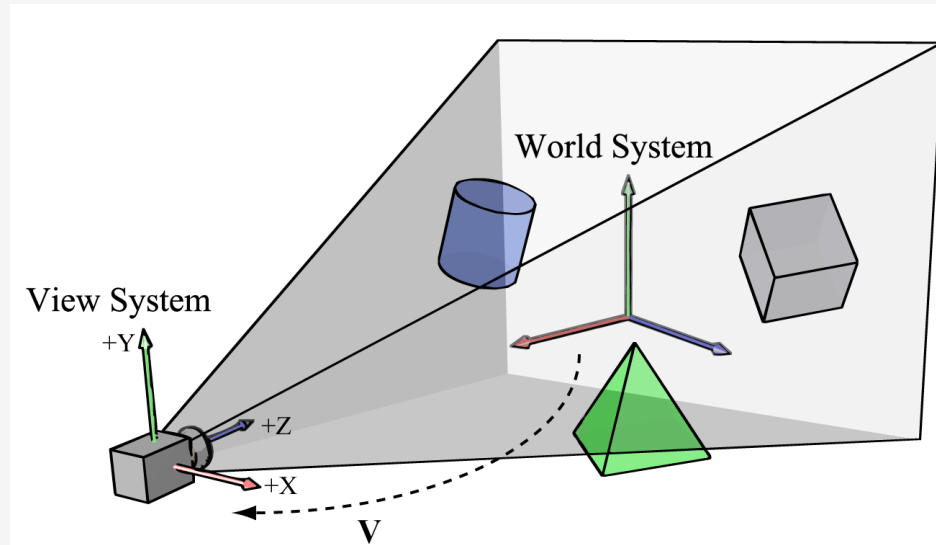
View Matrix

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:

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

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

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

THE CAMERA CLASS

To encapsulate our camera related code, we define and implement a Camera class.

The camera class stores:

1. The *position, right, up, and* 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.

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

```cpp
class Camera
{
public:

    Camera();
    ~Camera();
```



```cpp
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;
```
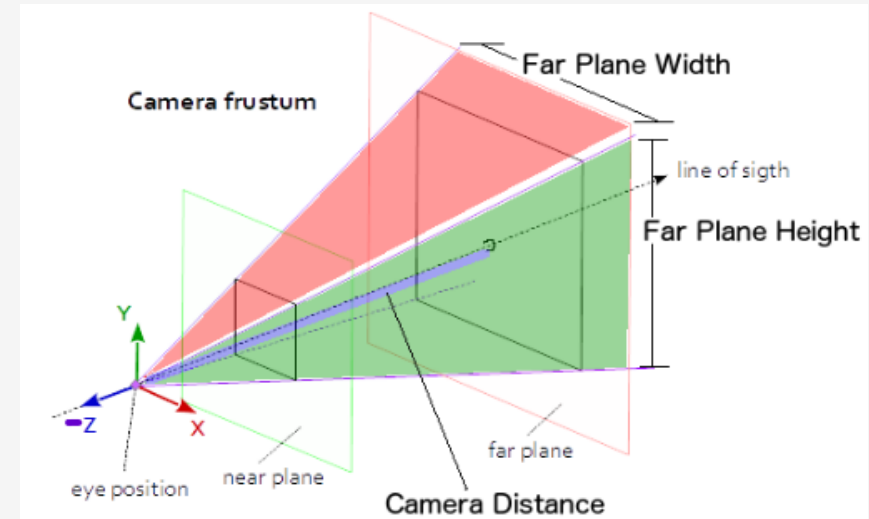
Camera::GetPosition & Camera::SetLens methods

We provide XMVECTOR return variations for many of the "get" methods;

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

```cpp
XMVECTOR Camera::GetPosition()const


{


return XMLoadFloat3(&mPosition);


}


XMFLOAT3 Camera::GetPosition3f()const


{


return mPosition;


}
```

We can think of the frustum as the lens of our camera, for it controls our view of view.

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

```cpp
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.

```cpp
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;
}
```
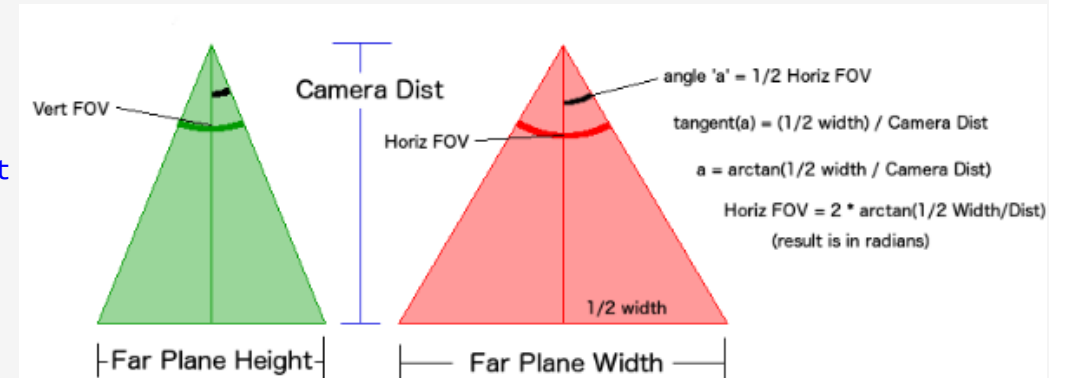
Transforming the Camera

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

1. Dollying – Walk method: 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. Trucking – Strafe method: Move the camera along its right vector to move(strafe) right and left. This can be implemented by translating the camera position along its right vector.

3. Tilting – Pitch method: 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. Panning – RotateY method: 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.`

5. Pedestal: Moving the camera vertically up or down along the world's *y*-axis (assuming the *y*-axis corresponds to the world's "up" direction). This can be implemented by translating the camera position along its up vector.

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);
```

The view matrix is column major. That is, in a 4×4 homogeneous transformation matrix, the first column represents the "right" vector, the second column represents the "up" vector, the third column represents the "forward" or "look" vector and the fourth column represents the translation vector (origin or position) of the space represented by the transformation matrix.

```
// 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;
}
}
```

$$\begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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:

```cpp
float mTheta = 1.5f*XM_PI;

float mPhi = 0.2f*XM_PI;

float mRadius = 15.0f;

Camera mCamera;
```

When the window is resized, we no longer rebuild the projection matrix explicitly, and instead delegate the work to the Camera class with SetLens:

```cpp
void CameraAndDynamicIndexingApp::OnResize()

{

    D3DApp::OnResize();

    // The window resized, update the aspect ratio and recompute the projection matrix.

        XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi, AspectRatio(),
            1.0f, 1000.0f);

        XMStoreFloat4x4(&mProj, P);

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

}
```

Update Scene: OnKeyboardInput method

```cpp
void StencilApp::OnKeyboardInput(const GameTimer& gt)
{ // Allow user to move skull.
const float dt = gt.DeltaTime();

if(GetAsyncKeyState('A') & 0x8000)
mSkullTranslation.x -= 1.0f*dt;

if(GetAsyncKeyState('D') & 0x8000)
mSkullTranslation.x += 1.0f*dt;

if(GetAsyncKeyState('W') & 0x8000)
mSkullTranslation.y += 1.0f*dt;

if(GetAsyncKeyState('S') & 0x8000)
mSkullTranslation.y -= 1.0f*dt;

// Don't let user move below ground plane.
mSkullTranslation.y = MathHelper::Max(mSkullTranslation.y, 0.0f);
// Update the new world matrix.
XMMATRIX skullRotate = XMMatrixRotationY(0.5f*MathHelper::Pi);
XMMATRIX skullScale = XMMatrixScaling(0.45f, 0.45f, 0.45f);
XMMATRIX skullOffset = XMMatrixTranslation(mSkullTranslation.x,
mSkullTranslation.y, mSkullTranslation.z);
XMMATRIX skullWorld = skullRotate*skullScale*skullOffset;
XMStoreFloat4x4(&mSkullRitem->World, skullWorld);
// Update reflection world matrix.
XMVECTOR mirrorPlane = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f); // xy
plane
//Builds a transformation matrix designed to reflect vectors
through a given plane.
XMMATRIX R = XMMatrixReflect(mirrorPlane);
XMStoreFloat4x4(&mReflectedSkullRitem->World, skullWorld * R);
mSkullRitem->NumFramesDirty = gNumFrameResources;
mReflectedSkullRitem->NumFramesDirty = gNumFrameResources;}
```

```cpp
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();

}
```

Update Scene: OnMouseMove methods

```cpp
void StencilApp::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 0.2 unit in the scene.
        float dx = 0.2f*static_cast<float>(x - mLastMousePos.x);

        float dy = 0.2f*static_cast<float>(y - mLastMousePos.y);

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

        // Restrict the radius.
        mRadius = MathHelper::Clamp(mRadius, 5.0f, 150.0f);

    }
    mLastMousePos.x = x;

    mLastMousePos.y = y;

}
```

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

```cpp
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:

```cpp
// Cache View/Proj matrices.
DirectX::XMFLOAT4X4 mView =
MathHelper::Identity4x4();

DirectX::XMFLOAT4X4 mProj =
MathHelper::Identity4x4();

// Strafe/Walk the camera a distance d.
void Strafe(float d);

void Walk(float d);

// Rotate the camera.
void Pitch(float angle);

void RotateY(float angle);

// After modifying camera position/orientation,
call to rebuild the view matrix.
void UpdateViewMatrix();
```
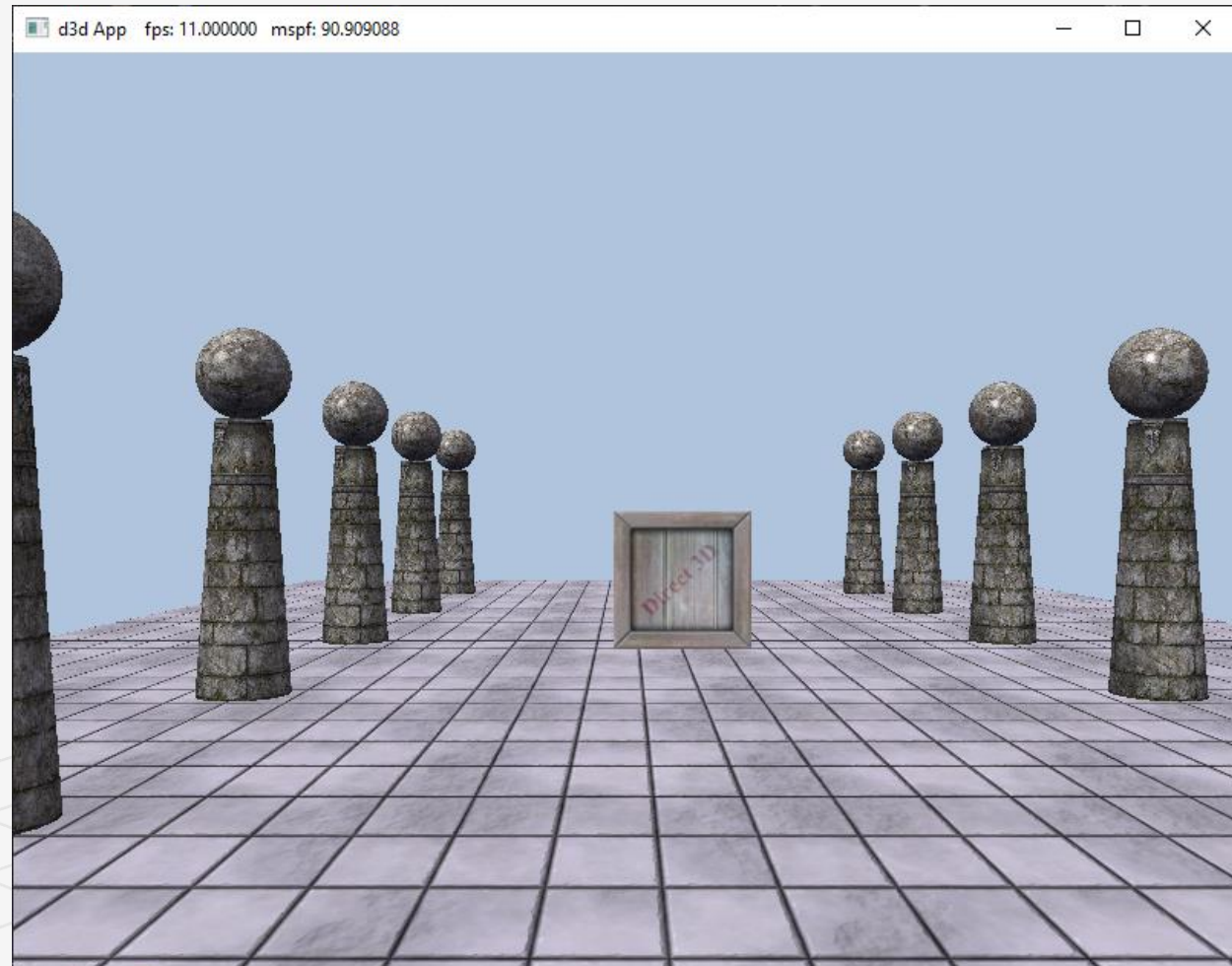
DYNAMIC INDEXING

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.

Create a structured buffer that stores all of the material data

Instead of storing our material data in ==constant buffers==, we will store it in a structured buffer. A structured buffer can be indexed in the shader program.

```
cbuffer cbMaterial : register(b2)


{


float4    gDiffuseAlbedo;


float3    gFresnelR0;


float     gRoughness;


float4x4 gMatTransform;


};
```

```
struct MaterialData


{


float4    DiffuseAlbedo;


float3    FresnelR0;


float     Roughness;


float4x4 MatTransform;


uint      DiffuseMapIndex;


uint      MatPad0;


uint      MatPad1;


uint      MatPad2;


};
```

Add a MaterialIndex field to our object constant buffer

```cpp
----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();

//Add a MaterialIndex field to our object constant
buffer to specify the index of the material  to use for
each draw call.

UINT DiffuseMapIndex = 0;

UINT MaterialPad0;
UINT MaterialPad1;
UINT MaterialPad2;
};
```

```hlsl
// Constant data that varies per frame.

cbuffer cbPerObject : register(b0)

{

float4x4 gWorld;

float4x4 gTexTransform;

uint gMaterialIndex;

uint gObjPad0;

uint gObjPad1;

uint gObjPad2;

};
```

ObjectConstants structure

We note that the ObjectConstants structure has been updated to have a MaterialIndex. The value you set for this is the same index you would have used to offset into the material constant buffer:

```
struct ObjectConstants

{

 DirectX::XMFLOAT4X4 World = MathHelper::Identity4x4();


DirectX::XMFLOAT4X4 TexTransform = MathHelper::Identity4x4();


UINT    MaterialIndex;


UINT    ObjPad0;


UINT    ObjPad1;


UINT    ObjPad2;

};
```

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

ObjectConstants objConstants;
XMStoreFloat4x4(&objConstants.World, XMMatrixTranspose(world));
XMStoreFloat4x4(&objConstants.TexTransform, XMMatrixTranspose(texTransform));
objConstants.MaterialIndex = e->Mat->MatCBIndex;

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

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

Bind *all* of the texture SRV descriptors used in the scene once per frame

```cpp
void TexColumnsApp::BuildRootSignature()

{

CD3DX12_DESCRIPTOR_RANGE texTable;

texTable.Init(

        D3D12_DESCRIPTOR_RANGE_TYPE_SRV,

        1,  // number of descriptors

        0); // register t0


    // Root parameter can be a table, root descriptor or root
constants.

    CD3DX12_ROOT_PARAMETER slotRootParameter[4];


// Perfomance TIP: Order from most frequent to least frequent.

slotRootParameter[0].InitAsDescriptorTable(1, &texTable,

D3D12_SHADER_VISIBILITY_PIXEL);

    slotRootParameter[1].InitAsConstantBufferView(0); // register b0

    slotRootParameter[2].InitAsConstantBufferView(1); // register b1

    slotRootParameter[3].InitAsConstantBufferView(2); // register b2
```

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

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

```cpp
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::DrawRenderItems

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

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

CD3DX12_GPU_DESCRIPTOR_HANDLE tex(mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());

tex.Offset(ri->Mat->DiffuseSrvHeapIndex, mCbvSrvDescriptorSize);

        D3D12_GPU_VIRTUAL_ADDRESS objCBAddress = objectCB->GetGPUVirtualAddress() + ri->ObjCBIndex*objCBByteSize;
        D3D12_GPU_VIRTUAL_ADDRESS matCBAddress = matCB->GetGPUVirtualAddress() + ri->Mat->MatCBIndex*matCBByteSize;

cmdList->SetGraphicsRootDescriptorTable(0, tex);
        cmdList->SetGraphicsRootConstantBufferView(1, objCBAddress);
        cmdList->SetGraphicsRootConstantBufferView(3, matCBAddress);
        cmdList->DrawIndexedInstanced(ri->IndexCount, 1, ri->StartIndexLocation, ri->BaseVertexLocation, 0);
    }
}
```

```cpp
void
CameraAndDynamicIndexingApp::DrawRenderItems(ID3D12GraphicsCom
mandList* 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;


cmdList->SetGraphicsRootConstantBufferView(0, objCBAddress);

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

CameraAndDynamicIndexingApp::Draw

```cpp
void TexColumnsApp::Draw(const GameTimer& gt)
{
    auto cmdListAlloc = mCurrFrameResource->CmdListAlloc;
    ThrowIfFailed(cmdListAlloc->Reset());
    ThrowIfFailed(mCommandList->Reset(cmdListAlloc.Get(),
    mPSOs["opaque"].Get()));
        mCommandList->RSSetViewports(1, &mScreenViewport);
        mCommandList->RSSetScissorRects(1, &mScissorRect);
    mCommandList->ResourceBarrier(1,
    &CD3DX12_RESOURCE_BARRIER::Transition(CurrentBackBuffer(),
    D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET));
    mCommandList->ClearRenderTargetView(CurrentBackBufferView(),
    Colors::LightSteelBlue, 0, nullptr);
        mCommandList->ClearDepthStencilView(DepthStencilView(),
    D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);
    mCommandList->OMSetRenderTargets(1, &CurrentBackBufferView(), true,
    &DepthStencilView());
    ID3D12DescriptorHeap* descriptorHeaps[] = { mSrvDescriptorHeap.Get() };
    mCommandList->SetDescriptorHeaps(_countof(descriptorHeaps), descriptorHeaps);
    mCommandList->SetGraphicsRootSignature(mRootSignature.Get());

    auto passCB = mCurrFrameResource->PassCB->Resource();
    mCommandList->SetGraphicsRootConstantBufferView(2, passCB->GetGPUVirtualAddress());

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

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:

```cpp
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);
```

The Vertex Shader Changes

```
Texture2D gDiffuseMap : register(t0);

VertexOut VS(VertexIn vin)

{

VertexOut vout = (VertexOut)0.0f;


// 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, gMatTransform).xy;

return vout;

}
```

```
// 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);


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;

}
```

The Pixel Shader Changes

```
float4 PS(VertexOut pin) : SV_Target
{
float4 diffuseAlbedo = gDiffuseMap.Sample(gsamLinearWrap,
pin.TexC) * gDiffuseAlbedo;

// Interpolating normal can unnormalize it, so renormalize it.
pin.NormalW = normalize(pin.NormalW);

// Vector from point being lit to eye.
float3 toEyeW = normalize(gEyePosW - pin.PosW);

// Light terms.
float4 ambient = gAmbientLight * diffuseAlbedo;

const float shininess = 1.0f - gRoughness;
Material mat = { diffuseAlbedo, gFresnelR0, shininess };
float3 shadowFactor = 1.0f;
float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
pin.NormalW, toEyeW, shadowFactor);

float4 litColor = ambient + directLight;

// Common convention to take alpha from diffuse albedo.
litColor.a = diffuseAlbedo.a;

return litColor;

}
```

```
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);
// Interpolating normal can unnormalize it, so renormalize it.
pin.NormalW = normalize(pin.NormalW);

// Vector from point being lit to eye.
float3 toEyeW = normalize(gEyePosW - pin.PosW);

// Light terms.
float4 ambient = gAmbientLight * diffuseAlbedo;

const float shininess = 1.0f - roughness;
Material mat = { diffuseAlbedo, fresnelR0, shininess };
float3 shadowFactor = 1.0f;
float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
pin.NormalW, toEyeW, shadowFactor);

float4 litColor = ambient + directLight;

// Common convention to take alpha from diffuse albedo.
litColor.a = diffuseAlbedo.a;

return litColor;
```

No more **UpdateMaterialCBs**

```cpp
void CameraAndDynamicIndexingApp::UpdateMaterialBuffer(const GameTimer& gt)

{
auto currMaterialBuffer = mCurrFrameResource->MaterialBuffer.get();

for(auto& e : mMaterials)

{
// Only update the cbuffer data if the constants have changed.  If the cbuffer

// data changes, it needs to be updated for each FrameResource.
Material* mat = e.second.get();

if(mat->NumFramesDirty > 0)
{
XMMATRIX matTransform = XMLoadFloat4x4(&mat->MatTransform);


MaterialData matData;

matData.DiffuseAlbedo = mat->DiffuseAlbedo;

matData.FresnelR0 = mat->FresnelR0;

matData.Roughness = mat->Roughness;

XMStoreFloat4x4(&matData.MatTransform, XMMatrixTranspose(matTransform));

matData.DiffuseMapIndex = mat->DiffuseSrvHeapIndex;


currMaterialBuffer->CopyData(mat->MatCBIndex, matData);


// Next FrameResource need to be updated too.

mat->NumFramesDirty--;

}
}
}
```

```cpp
void CameraAndDynamicIndexingApp::UpdateMaterialCBs(const GameTimer& gt)
{
auto currMaterialCB = mCurrFrameResource->MaterialCB.get();
for (auto& e : mMaterials)
{
// Only update the cbuffer data if the constants have changed.  If the cbuffer
// data changes, it needs to be updated for each FrameResource.
Material* mat = e.second.get();
if (mat->NumFramesDirty > 0)
{
XMMATRIX matTransform = XMLoadFloat4x4(&mat->MatTransform);

MaterialConstants matConstants;
matConstants.DiffuseAlbedo = mat->DiffuseAlbedo;
matConstants.FresnelR0 = mat->FresnelR0;
matConstants.Roughness = mat->Roughness;
XMStoreFloat4x4(&matConstants.MatTransform,
XMMatrixTranspose(matTransform));

currMaterialCB->CopyData(mat->MatCBIndex, matConstants);

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

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.