Game Engine Development II

Week3

lamat

Hooman Salamat

DYNAMIC CUBE MAPS

So far we have described static cube maps, where the images stored in the cube map are premade and fixed.

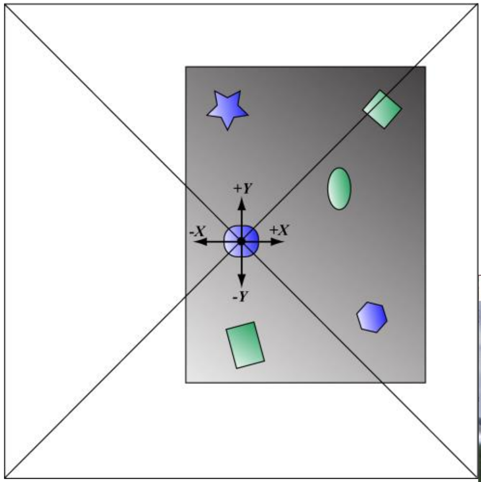
With a pre-generated cube map, you cannot capture animated objects, which means we cannot reflect animated objects.

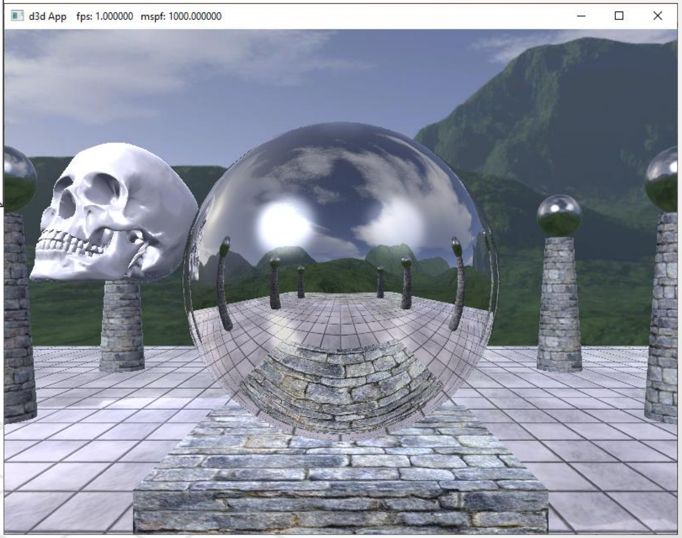
To overcome this limitation, we can build the cube map at runtime.

Every frame you position the camera in the scene that is to be the origin of the cube map, and then *render the scene six times into each cube map face* along each coordinate axis direction.

The camera is placed at position *O* in the scene, centered about the object we want to generate the dynamic cube map relative to.

We render the scene six times along each coordinate axis direction with a field of view angle of 90° so that the image of the entire surrounding environment is captured.





Dynamic Cube Map Helper Class

To help render to a cube map dynamically, we create the following CubeRenderTarget class, which encapsulates the actual ID3D12Resource object of the cube map, the various descriptors to the resource, and other useful data for rendering to the cube map:

```
class CubeRenderTarget
{
public:
CubeRenderTarget(ID3D12Device* device,
UINT width, UINT height,
DXGI_FORMAT format);

CubeRenderTarget(const CubeRenderTarget& rhs)=delete;
CubeRenderTarget& operator=(const CubeRenderTarget& rhs)=delete;
~CubeRenderTarget()=default;

ID3D12Resource* Resource();
CD3DX12_GPU_DESCRIPTOR_HANDLE Srv();
CD3DX12_CPU_DESCRIPTOR_HANDLE Rtv(int faceIndex);

D3D12_VIEWPORT Viewport()const;
D3D12_RECT ScissorRect()const;

void BuildDescriptors(
CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuSrv,
CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuSrv,
CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuRtv[6]);

void OnResize(UINT newWidth, UINT newHeight);
```

```
private:
void BuildDescriptors();
void BuildResource();

private:

ID3D12Device* md3dDevice = nullptr;

D3D12_VIEWPORT mViewport;
D3D12_RECT mScissorRect;

UINT mWidth = 0;
UINT mHeight = 0;
DXGI_FORMAT mFormat = DXGI_FORMAT_R8G8B8A8_UNORM;

CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuSrv;
CD3DX12_GPU_DESCRIPTOR_HANDLE mhGpuSrv;
CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuRtv[6];

Microsoft::WRL::ComPtr<ID3D12Resource> mCubeMap =
nullptr;
};
```

Building the Cube Map Resource

Creating a cube map texture is done by creating a texture array with six elements (one for each face).

Because we are going to render to the cube map, we must set the D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET flag.

The following method builds the cube map resource:

```
void CubeRenderTarget::BuildResource()
{

D3D12_RESOURCE_DESC texDesc;
ZeroMemory(&texDesc, sizeof(D3D12_RESOURCE_DESC));
texDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
texDesc.Alignment = 0;
texDesc.Width = mWidth;
texDesc.Height = mHeight;
texDesc.DepthOrArraySize = 6;
texDesc.MipLevels = 1;
texDesc.Format = mFormat;
texDesc.SampleDesc.Count = 1;
texDesc.SampleDesc.Quality = 0;
texDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
texDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET;

ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
D3D12_HEAP_FLAG_NONE,
&texDesc,
D3D12_RESOURCE_STATE_GENERIC_READ,
nullptr,
IID_PPV_ARGS(&mCubeMap)));
}
```

Extra Descriptor Heap Space

Rendering to a cube map requires six additional render target views, one for each face, and one additional depth/stencil buffer.

Therefore, we must override the D3DApp::CreateRtvAndDsvDescriptorHeaps method and allocate these additional descriptors:

```
void DynamicCubeMapApp::CreateRtvAndDsvDescriptorHeaps()
{
// Add +6 RTV for cube render target.
D3D12_DESCRIPTOR_HEAP_DESC rtvHeapDesc;
rtvHeapDesc.NumDescriptors = SwapChainBufferCount + 6;
rtvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
rtvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
rtvHeapDesc.NodeMask = 0;
ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
&rtvHeapDesc, IID_PPV_ARGS(mRtvHeap.GetAddressOf())));

// Add +1 DSV for cube render target.
D3D12_DESCRIPTOR_HEAP_DESC dsvHeapDesc;
dsvHeapDesc.NumDescriptors = 2;
dsvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_DSV;
dsvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
dsvHeapDesc.NodeMask = 0;
ThrowIfFailed(md3dDevice->CreateDescriptorHeap(
&dsvHeapDesc, IID_PPV_ARGS(mDsvHeap.GetAddressOf())));

mCubeDSV = CD3DX12_CPU_DESCRIPTOR_HANDLE(
mDsvHeap->GetCPUDescriptorHandleForHeapStart(),
1,
mDsvDescriptorSize);
}
```

DynamicCubeMapApp::BuildDescriptorHeaps

In addition, we will need one extra SRV so that we can bind the cube map as a shader input after it has been generated.

The descriptor handles are passed into the CubeRenderTarget::BuildDescriptors method which saves a copy of the handles and then actually creates the views:

```
auto srvCpuStart = mSrvDescriptorHeap->GetCPUDescriptorHandleForHeapStart();
auto srvGpuStart = mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart();
auto rtvCpuStart = mRtvHeap->GetCPUDescriptorHandleForHeapStart();

// Cubemap RTV goes after the swap chain descriptors.
int rtvOffset = SwapChainBufferCount;

CD3DX12_CPU_DESCRIPTOR_HANDLE cubeRtvHandles[6];
for(int i = 0; i < 6; ++i)
cubeRtvHandles[i] = CD3DX12_CPU_DESCRIPTOR_HANDLE(rtvCpuStart, rtvOffset + i,
mRtvDescriptorSize);

// Dynamic cubemap SRV is created after the sky SRV.
mDynamicCubeMap->BuildDescriptors(
CD3DX12_CPU_DESCRIPTOR_HANDLE(srvCpuStart, mDynamicTexHeapIndex, mCbvSrvUavDescriptorSize),
CD3DX12_GPU_DESCRIPTOR_HANDLE(srvGpuStart, mDynamicTexHeapIndex, mCbvSrvUavDescriptorSize),
cubeRtvHandles);
```

Building the Descriptors

We now need to create an SRV to the cube map resource so that we can sample it in a pixel shader after it is built.

We also need to create a render target view to each element in the cube map texture array, so that we can render onto each cube map face one by- one.

The following method creates the necessary views:

```
void CubeRenderTarget::BuildDescriptors()
{
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = mFormat;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURECUBE;
srvDesc.TextureCube.MostDetailedMip = 0;
srvDesc.TextureCube.MipLevels = 1;
srvDesc.TextureCube.ResourceMinLODClamp = 0.0f;

// Create SRV to the entire cubemap resource.
md3dDevice->CreateShaderResourceView(mCubeMap.Get(), &srvDesc, mhCpuSrv);

// Create RTV to each cube face.
for(int i = 0; i < 6; ++i)
{
D3D12_RENDER_TARGET_VIEW_DESC rtvDesc;
rtvDesc.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2DARRAY;
rtvDesc.Format = mFormat;
rtvDesc.Texture2DArray.MipSlice = 0;
rtvDesc.Texture2DArray.PlaneSlice = 0;

// Render target to ith element.
rtvDesc.Texture2DArray.FirstArraySlice = i;

// Only view one element of the array.
rtvDesc.Texture2DArray.ArraySize = 1;

// Create RTV to ith cubemap face.
md3dDevice->CreateRenderTargetView(mCubeMap.Get(), &rtvDesc, mhCpuRtv[i]);
}
}
```

Building the Depth Buffer

Because we render to the cube faces one at a time, we only need one depth buffer for the cube map rendering.

We build an additional depth buffer and DSV with the following code:

```
void DynamicCubeMapApp::BuildCubeDepthStencil()
{
// Create the depth/stencil buffer and view.
D3D12_RESOURCE_DESC depthStencilDesc;
depthStencilDesc.Dimension = D3D12_RESOURCE_DIMENSION_TEXTURE2D;
depthStencilDesc.Alignment = 0;
depthStencilDesc.Width = CubeMapSize;
depthStencilDesc.Height = CubeMapSize;
depthStencilDesc.DepthOrArraySize = 1;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.Format = mDepthStencilFormat;
depthStencilDesc.SampleDesc.Count = 1;
depthStencilDesc.SampleDesc.Quality = 0;
depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;

D3D12_CLEAR_VALUE optClear;
optClear.Format = mDepthStencilFormat;
optClear.DepthStencil.Depth = 1.0f;
optClear.DepthStencil.Stencil = 0;
ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
D3D12_HEAP_FLAG_NONE,
&depthStencilDesc,
D3D12_RESOURCE_STATE_COMMON,
&optClear,
IID_PPV_ARGS(mCubeDepthStencilBuffer.GetAddressOf())));

// Create descriptor to mip level 0 of entire resource using the format of the resource.
md3dDevice->CreateDepthStencilView(mCubeDepthStencilBuffer.Get(), nullptr, mCubeDSV);

// Transition the resource from its initial state to be used as a depth buffer.
mCommandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(mCubeDepthStencilBuffer.Get(),
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_DEPTH_WRITE));
}
```

Cube Map Viewport and Scissor Rectangle

Because the cube map faces will have a different resolution than the main back buffer, we need to define a new viewport and scissor rectangle that covers a cube map face:

```
CubeRenderTarget::CubeRenderTarget(ID3D12Device* device,
                                   UINT width, UINT height,
                                       DXGI_FORMAT format)
{
md3dDevice = device;

mWidth = width;
mHeight = height;
mFormat = format;

mViewport = { 0.0f, 0.0f, (float)width, (float)height, 0.0f, 1.0f };
mScissorRect = { 0, 0, (int)width, (int)height };

BuildResource();
}

D3D12_VIEWPORT CubeRenderTarget::Viewport()const
{
return mViewport;
}

D3D12_RECT CubeRenderTarget::ScissorRect()const
{
return mScissorRect;
}
```

Setting up the Cube Map Camera

To generate a cube map idea is to position a camera at the center of some object *O* in the scene with a 90° field of view angle (both vertically and horizontally).

Then have the camera look down the positive *x*-axis, negative *x*-axis, positive *y*-axis, negative *y*-axis, positive *z*-axis, and negative *z*-axis, and to take a picture of the scene (excluding the object *O*) from each of these six viewpoints.

To facilitate this, we generate six cameras, one for each face, centered at the given position (*x, y, z*):

```
void DynamicCubeMapApp::BuildCubeFaceCamera(float x, float y, float z)
{
// Generate the cube map about the given position.
XMFLOAT3 center(x, y, z);
XMFLOAT3 worldUp(0.0f, 1.0f, 0.0f);

// Look along each coordinate axis.
XMFLOAT3 targets[6] =
{
XMFLOAT3(x + 1.0f, y, z), // +X
XMFLOAT3(x - 1.0f, y, z), // -X
XMFLOAT3(x, y + 1.0f, z), // +Y
XMFLOAT3(x, y - 1.0f, z), // -Y
XMFLOAT3(x, y, z + 1.0f), // +Z
XMFLOAT3(x, y, z - 1.0f)  // -Z
};

// Use world up vector (0,1,0) for all directions except +Y/-Y.  In these cases, we
// are looking down +Y or -Y, so we need a different "up" vector.
XMFLOAT3 ups[6] =
{
XMFLOAT3(0.0f, 1.0f, 0.0f),  // +X
XMFLOAT3(0.0f, 1.0f, 0.0f),  // -X
XMFLOAT3(0.0f, 0.0f, -1.0f), // +Y
XMFLOAT3(0.0f, 0.0f, +1.0f), // -Y
XMFLOAT3(0.0f, 1.0f, 0.0f), // +Z
XMFLOAT3(0.0f, 1.0f, 0.0f) // -Z
};

for(int i = 0; i < 6; ++i)
{
mCubeMapCamera[i].LookAt(center, targets[i], ups[i]);
mCubeMapCamera[i].SetLens(0.5f*XM_PI, 1.0f, 0.1f, 1000.0f);
mCubeMapCamera[i].UpdateViewMatrix();
}
}
```

DynamicCubeMapApp::BuildFrameResources

Because rendering to each cube map face utilizes a different camera, each cube face needs its own set of PassConstants, we need to increase our PassConstants count by six when we create our frame resources.

Element 0 will correspond to our main rendering pass, and elements 1-6 will correspond to our cube map faces.

```
void DynamicCubeMapApp::BuildFrameResources()
{
    for(int i = 0; i < gNumFrameResources; ++i)
    {
mFrameResources.push_back(std::make_unique<FrameResource>(md3dDevice.Get(),
            7, (UINT)mAllRitems.size(), (UINT)mMaterials.size()));
    }
}
```

DynamicCubeMapApp::UpdateCubeMapFacePassCBs

We implement the following method to set the constant data for each cube map face:

```
void DynamicCubeMapApp::UpdateCubeMapFacePassCBs()
{
for(int i = 0; i < 6; ++i)
{
PassConstants cubeFacePassCB = mMainPassCB;

XMMATRIX view = mCubeMapCamera[i].GetView();
XMMATRIX proj = mCubeMapCamera[i].GetProj();

XMMATRIX viewProj = XMMatrixMultiply(view, proj);
XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(view), view);
XMMATRIX invProj = XMMatrixInverse(&XMMatrixDeterminant(proj), proj);
XMMATRIX invViewProj = XMMatrixInverse(&XMMatrixDeterminant(viewProj), viewProj);

XMStoreFloat4x4(&cubeFacePassCB.View, XMMatrixTranspose(view));
XMStoreFloat4x4(&cubeFacePassCB.InvView, XMMatrixTranspose(invView));
XMStoreFloat4x4(&cubeFacePassCB.Proj, XMMatrixTranspose(proj));
XMStoreFloat4x4(&cubeFacePassCB.InvProj, XMMatrixTranspose(invProj));
XMStoreFloat4x4(&cubeFacePassCB.ViewProj, XMMatrixTranspose(viewProj));
XMStoreFloat4x4(&cubeFacePassCB.InvViewProj, XMMatrixTranspose(invViewProj));
cubeFacePassCB.EyePosW = mCubeMapCamera[i].GetPosition3f();
cubeFacePassCB.RenderTargetSize = XMFLOAT2((float)CubeMapSize, (float)CubeMapSize);
cubeFacePassCB.InvRenderTargetSize = XMFLOAT2(1.0f / CubeMapSize, 1.0f / CubeMapSize);

auto currPassCB = mCurrFrameResource->PassCB.get();

// Cube map pass cbuffers are stored in elements 1-6.
currPassCB->CopyData(1 + i, cubeFacePassCB);
}
}
```

Drawing into the Cube Map

We have three render layers:

```
enum class RenderLayer : int

{

Opaque = 0,

OpaqueDynamicReflectors,

Sky,

Count

};
```

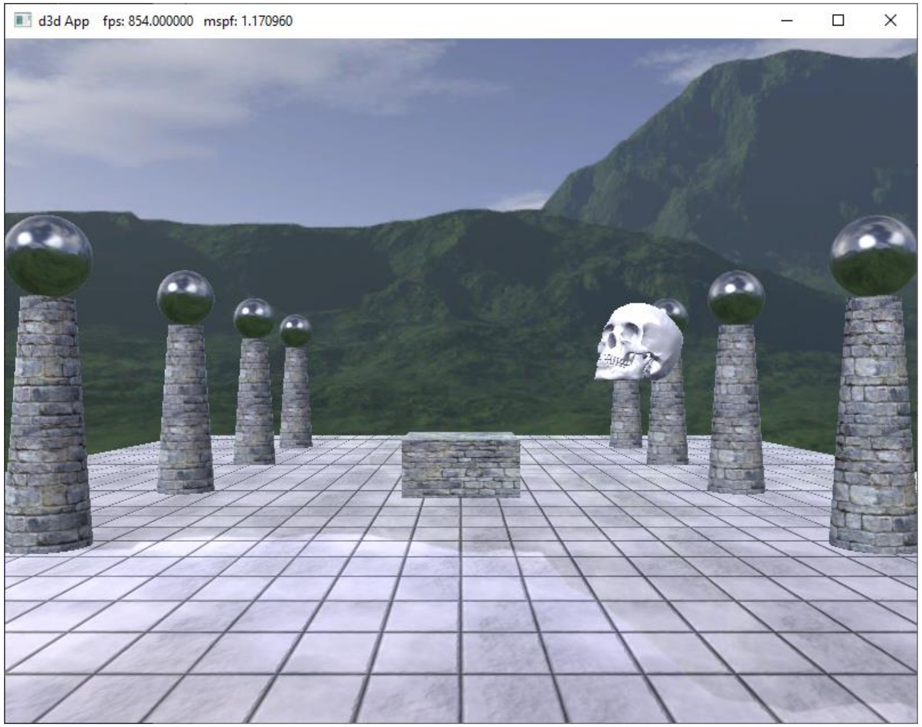
The OpaqueDynamicReflectors layer contains the center sphere which will use the dynamic cube map to reflect local dynamic objects.

Our first step is to draw the scene to each face of the cube map, but not including the center sphere; this means we just need to render the opaque and sky layers to the cube map.

Important: At this point, we are not showing anything on the display. We just "rendered" our environment (next slide) to a cube map which is a resource of texture cube type!

```
void DynamicCubeMapApp::DrawSceneToCubeMap()
{
mCommandList->RSSetViewports(1, &mDynamicCubeMap->Viewport());
mCommandList->RSSetScissorRects(1, &mDynamicCubeMap->ScissorRect());

// Change to RENDER_TARGET.
mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mDynamicCubeMap->Resource(),
D3D12_RESOURCE_STATE_GENERIC_READ, D3D12_RESOURCE_STATE_RENDER_TARGET));

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

// For each cube map face.
for(int i = 0; i < 6; ++i)
{
// Clear the back buffer and depth buffer.
mCommandList->ClearRenderTargetView(mDynamicCubeMap->Rtv(i), Colors::LightSteelBlue, 0, nullptr);
mCommandList->ClearDepthStencilView(mCubeDSV, D3D12_CLEAR_FLAG_DEPTH | D3D12_CLEAR_FLAG_STENCIL, 1.0f, 0, 0, nullptr);

// Specify the buffers we are going to render to.
mCommandList->OMSetRenderTargets(1, &mDynamicCubeMap->Rtv(i), true, &mCubeDSV);

// Bind the pass constant buffer for this cube map face so we use
// the right view/proj matrix for this cube face.
auto passCB = mCurrFrameResource->PassCB->Resource();
D3D12_GPU_VIRTUAL_ADDRESS passCBAddress = passCB->GetGPUVirtualAddress() + (1+i)*passCBByteSize;
mCommandList->SetGraphicsRootConstantBufferView(1, passCBAddress);

DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque]);

mCommandList->SetPipelineState(mPSOs["sky"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Sky]);

mCommandList->SetPipelineState(mPSOs["opaque"].Get());
}

// Change back to GENERIC_READ so we can read the texture in a shader.
mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mDynamicCubeMap->Resource(),
D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_GENERIC_READ));
}
```

Draw the scene to each face of the cube map, but not including the center sphere



DynamicCubeMapApp::Draw

Finally, after we rendered the scene to the cube map, we set our main render targets and draw the center sphere with the dynamic cube map applied to it, and then draw the scene and the sky:

```
DrawSceneToCubeMap();

mCommandList->RSSetViewports(1, &mScreenViewport);
mCommandList->RSSetScissorRects(1, &mScissorRect);

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

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

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

auto passCB = mCurrFrameResource->PassCB->Resource();
mCommandList->SetGraphicsRootConstantBufferView(1, passCB->GetGPUVirtualAddress());


// Use the dynamic cube map for the dynamic reflectors layer.
CD3DX12_GPU_DESCRIPTOR_HANDLE dynamicTexDescriptor(mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());
dynamicTexDescriptor.Offset(mSkyTexHeapIndex + 1, mCbvSrvUavDescriptorSize);
mCommandList->SetGraphicsRootDescriptorTable(3, dynamicTexDescriptor);

DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::OpaqueDynamicReflectors]);

// Use the static "background" cube map for the other objects (including the sky)
mCommandList->SetGraphicsRootDescriptorTable(3, skyTexDescriptor);

DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Opaque]);

mCommandList->SetPipelineState(mPSOs["sky"].Get());
DrawRenderItems(mCommandList.Get(), mRitemLayer[(int)RenderLayer::Sky]);

    // 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));
```

DYNAMIC CUBE MAPS WITH THE GEOMETRY SHADER

In the previous slide, we redrew the scene six times to generate the cube map—once for each cube map face.

You also could use the geometry shader to render a cube map by drawing the scene only once.

First, it creates a render target view to the *entire* texture array (not each individual face texture):

```
// Create the 6-face render target view

D3D12_RENDER_TARGET_VIEW_DESC rtvDesc;
rtvDesc.ViewDimension = D3D12_RTV_DIMENSION_TEXTURE2DARRAY;
rtvDesc.Format = mFormat;
rtvDesc.Texture2DArray.MipSlice = 0;
rtvDesc.Texture2DArray.PlaneSlice = 0;

// Render target to ith element.
rtvDesc.Texture2DArray.FirstArraySlice = 0;

// view six elements of the array.
rtvDesc.Texture2DArray.ArraySize = 6;

// Create RTV to ith cubemap face.
md3dDevice->CreateRenderTargetView(mCubeMap.Get(), &rtvDesc, &mhCpuRtv);
```

Create the depth stencil view for the entire cube

CubeMapGS requires a cube map of depth buffers (one for each face). The depth stencil view to the *entire* texture array of depth buffers creates as follows:

It then binds this render target and depth stencil view to the OM stage of the pipeline:

we have bound a view to an array of render targets and a view to an array of depth stencil buffers to the OM stage, and we are going to render to each array slice simultaneously.

```
void DynamicCubeMapApp::BuildCubeDepthStencil()
{
// Create the depth/stencil buffer and view.
D3D12_RESOURCE_DESC depthStencilDesc;
depthStencilDesc.Dimension = D3D12_DSV_DIMENSION_TEXTURE2DARRAY;
depthStencilDesc.Alignment = 0;
depthStencilDesc.Width = CubeMapSize;
depthStencilDesc.Height = CubeMapSize;
depthStencilDesc.DepthOrArraySize = 6;
depthStencilDesc.MipLevels = 1;
depthStencilDesc.Format = mDepthStencilFormat;
depthStencilDesc.SampleDesc.Count = 1;
depthStencilDesc.SampleDesc.Quality = 0;
depthStencilDesc.Layout = D3D12_TEXTURE_LAYOUT_UNKNOWN;
depthStencilDesc.Flags = D3D12_RESOURCE_FLAG_ALLOW_DEPTH_STENCIL;

depthStencilDesc.Texture2DArray.FirstArraySlice = 0;

depthStencilDesc.Texture2DArray.ArraySize = 6;

depthStencilDesc.Texture2DArray.MipSlice = 0;



D3D12_CLEAR_VALUE optClear;
optClear.Format = mDepthStencilFormat;
optClear.DepthStencil.Depth = 1.0f;
optClear.DepthStencil.Stencil = 0;
ThrowIfFailed(md3dDevice->CreateCommittedResource(
&CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_DEFAULT),
D3D12_HEAP_FLAG_NONE,
&depthStencilDesc,
D3D12_RESOURCE_STATE_COMMON,
&optClear,
IID_PPV_ARGS(mCubeDepthStencilBuffer.GetAddressOf())));

// Create descriptor to mip level 0 of entire resource using the format of the resource.
md3dDevice->CreateDepthStencilView(mCubeDepthStencilBuffer.Get(), nullptr, mCubeDSV);

// Transition the resource from its initial state to be used as a depth buffer.
mCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(mCubeDepthStencilBuffer.Get(),
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_DEPTH_WRITE));
```

GS_CubeMap

Now, the scene is rendered once and an array of six view matrices is available in the constant buffers.

The geometry shader replicates the input triangle six times, and assigns the triangle to one of the six render target array slices.

Assigning a triangle to a render target array slice is done by setting the system value SV_RenderTargetArrayIndex.

This system value is an integer index value that can only be set as an output from the geometry shader to specify the index of the render target array slice the primitive should be rendered onto.

This system value can only be used if the render target view is actually a view to an array resource.

```
struct PS_CUBEMAP_IN
{
float4 Pos : SV_POSITION; // Projection coord
float2 Tex : TEXCOORD0; // Texture coord
uint RTIndex : SV_RenderTargetArrayIndex;
};
[maxvertexcount(18)]
void GS_CubeMap(triangle GS_CUBEMAP_IN input[3],
inout TriangleStream<PS_CUBEMAP_IN> CubeMapStream)
{
// For each triangle
for (int f = 0; f < 6; ++f)
{
// Compute screen coordinates
PS_CUBEMAP_IN output;
// Assign the ith triangle to the ith render target.
output.RTIndex = f;
// For each vertex in the triangle
for (int v = 0; v < 3; v++)
{
// Transform to the view space of the ith cube face.

output.Pos = mul(input[v].Pos, g_mViewCM[f]);
// Transform to homogeneous clip space.
output.Pos = mul(output.Pos, mProj);
output.Tex = input[v].Tex;
CubeMapStream.Append(output);
}
CubeMapStream.RestartStrip();
}
}
```

Conclusion

- This strategy is interesting and demonstrates simultaneous render targets and the SV_RenderTargetArrayIndex system value; however, it is not a definite win. There are two issues that make this method unattractive:
- 1. It uses the geometry shader to output a large set of data. We mentioned the geometry shader acts inefficiently when outputting a large set of data. Therefore, using a geometry shader for this purpose could hurt performance.
- 2. In a typical scene, a triangle will not overlap more than one cube map . Therefore, the act of replicating a triangle and rendering it onto each cube face when it will be clipped by five out of six of the faces is wasteful.
- 3. In real applications (non-demo), we would use frustum culling , and only render the objects visible to a particular cube map face. Frustum culling at the object level cannot be done by a geometry shader implementation.
- 4. On the other hand, a situation where this strategy does work well would be rendering a mesh that surrounds the scene. For example, suppose that you had a dynamic sky system where the clouds moved and the sky color changed based on the time of day. Because the sky is changing, we cannot use a premade cube map texture to reflect the sky, so we have to use a dynamic cube map. Since the sky mesh surrounds the entire scene, it *is visible by all six* cube map faces Therefore, the second bullet point above does not apply, and the geometry shader method could be a win by reducing draw calls from six to one, assuming usage of the geometry shader does not hurt performance too much.

Scene Rendering

Objectives

- Examine entity systems in concept and practice
- Explore the viewable area of our world and scrolling
- Implement tree-based scene graphs, rendering and updating of many entities
- Implement the composition of all elements to shape the world

The `Entity` Class

- An entity represents some game element in the world
  - Other planes (friendly and enemy)
  - Projectiles (bullets and missiles)
  - Pickups
- Basically what the player can interact with
- Going to have a `velocity` attribute
- Let’s see what the definition looks like...

The Entity Class

```
class Entity
{
public:
   void setVelocity(sf::Vector2f velocity);
   void setVelocity(float vx, float vy);
   sf::Vector2f getVelocity() const;
private:
   sf::Vector2f mVelocity;
};
```

- Vector2f has a default constructor that sets x and y to zero

The Entity Class

```
void Entity::setVelocity(sf::Vector2f velocity)
{
   mVelocity = velocity;
}

void Entity::setVelocity(float vx, float vy)
{
   mVelocity.x = vx;
   mVelocity.y = vy;
}

sf::Vector2f Entity::getVelocity() const
{
   return mVelocity;
}
```

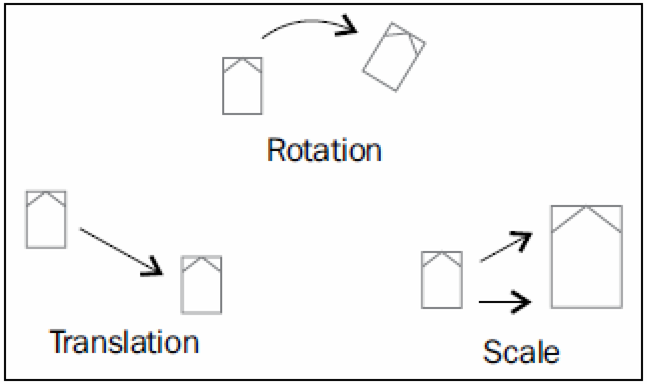
Aircraft Class

```
class Aircraft : public Entity
{
public:
    enum Type
    {
      Eagle,
      Raptor,
    };
public:
    explicit Aircraft(Type type);
private:
    Type mType;
};
```

Transforms

- A geometrical transform specifies the way an object is represented on screen
  - Translation -> position
  - Rotation -> orientation
  - Scale -> size

- SFML provides these in a class called sf::Transformable



sf::Transformable

- Accessors (getters/setters):
  - setPosition (), move(), rotate(), getScale()
  - setOrigin(), getOrigin()

- High-level classes such as `Sprite`, `Text` and `Shape` are derived from `Transformable` and `Drawable`

Sf::Drawable

- sf::Drawable is a stateless interface and provides only a pure virtual function with the following signature:

```
virtual void Drawable::draw(sf::RenderTarget& target,sf::RenderStates states) const = 0
```

- The first parameter specifies, where the drawable object is drawn to. Mostly, this will be a sf::RenderWindow. The second parameter contains additional information for the rendering process, such as blend mode (how pixel of the object are blened, transform (how the object is positioned/rotated/scaled), the used texture (what image is mapped to the object), or shader (what custom effectis applied to the object).
- SFML's high-level classes Sprite, Text, and Shape are all derived from Transformable and Drawable interfaces.

Scene Graphs

- A scene graph is developed to transform the hierarchies
  - Consists of multiple nodes called scene nodes
  - Each node can store an object that is drawn
  - Represented by class called `SceneNode`
  - To store the children, we use vector container `std::vector<SceneNode>`

SceneNode Class

```
class SceneNode
{
public:
  typedef std::unique_ptr<SceneNode> Ptr;
public:
  SceneNode();
private:
  std::vector<Ptr> mChildren;
  SceneNode* mParent;
};
```

SceneNode Class

- We provide an interface to insert and remove child nodes:

```
void attachChild(Ptr child);
Ptr detachChild(const SceneNode& node);
```

SceneNode Class

```
void SceneNode::attachChild(Ptr child)
{
    child->mParent = this;
    mChildren.push_back(std::move(child));
}

SceneNode::Ptr SceneNode::detachChild(const SceneNode& node)
{
   auto found = std::find_if(mChildren.begin(), mChildren.end(),
   [&] (Ptr& p) -> bool { return p.get() == &node; });

   assert(found != mChildren.end());
   Ptr result = std::move(*found);
   result->mParent = nullptr;
   mChildren.erase(found);
   return result;
}
```

SceneNode Class Updated

```
class SceneNode : public sf::Transformable, public sf::Drawable,
                    private sf::NonCopyable
{
public:
   typedef std::unique_ptr<SceneNode> Ptr;
public:
   SceneNode();
   void attachChild(Ptr child);
   Ptr detachChild(const SceneNode& node);
private:
   virtual void draw(sf::RenderTarget& target,
                       sf::RenderStates states) const;
   virtual void drawCurrent(sf::RenderTarget& target,
                              sf::RenderStates states) const;
private:
   std::vector<Ptr> mChildren;
   SceneNode* mParent;
};
```

SceneNode Class Updated

- The class can then be used thus:

```
sf::RenderWindow window(...);
SceneNode::Ptr node(...);
window.draw(*node); // note: no node->draw(window) here!
```

Aircraft Revisited

```
class Aircraft : public Entity // inherits indirectly SceneNode
{
public:
   explicit Aircraft(Type type);
   virtual void drawCurrent(sf::RenderTarget& target,
                            sf::RenderStates states) const;
private:
   Type mType;
   sf::Sprite mSprite;
};

void Aircraft::drawCurrent(sf::RenderTarget& target,
                            sf::RenderStates states) const
{
   target.draw(mSprite, states);
}
```

Resetting the Origin

- By default, the origin of sprites is in their upper-left corner
  - For alignment or rotation, it might be better to work with their center, and we can set it thus:

```
sf::FloatRect bounds = mSprite.getLocalBounds();
mSprite.setOrigin(bounds.width / 2.f, bounds.height / 2.f);
```

Scene Layers

- Different nodes must be rendered in a certain order
  - Can't have ground above sky, for example
  - Common sense
  - UI as top layer

```
enum Layer
{
   Background,
   Air,
   LayerCount
};
```

Updating the Scene

- During an update, entities move and interact, collisions are checked and projectiles are launched
- We can add the following to `SceneNode`

```
public:
  void update(sf::Time dt);
private:
  virtual void updateCurrent(sf::Time dt);
  void updateChildren(sf::Time dt);
```

Updating the Scene

```
void SceneNode::update(sf::Time dt)
{
   updateCurrent(dt);
   updateChildren(dt);
}

void SceneNode::updateCurrent(sf::Time)
{
}

void SceneNode::updateChildren(sf::Time dt)
{
   FOREACH(Ptr& child, mChildren)
       child->update(dt);
}
```

Updating the Scene

- We also have to make the following additions to the `Entity` class:

```
private:
   virtual void updateCurrent(sf::Time dt);

...

void Entity::updateCurrent(sf::Time dt)
{
   move(mVelocity * dt.asSeconds());
}
```

Absolute Transforms

- In order to find out if two objects collide, we have to look at their world transform, not local or relative transforms
- We perform the following absolute transform:

```
sf::Transform SceneNode::getWorldTransform() const
{
   sf::Transform transform = sf::Transform::Identity;
   for (const SceneNode* node = this; node != nullptr;
        node = node->mParent)
        transform = node->getTransform() * transform;
   return transform;
}

sf::Vector2f SceneNode::getWorldPosition() const
{
   return getWorldTransform() * sf::Vector2f();
}
```

The View

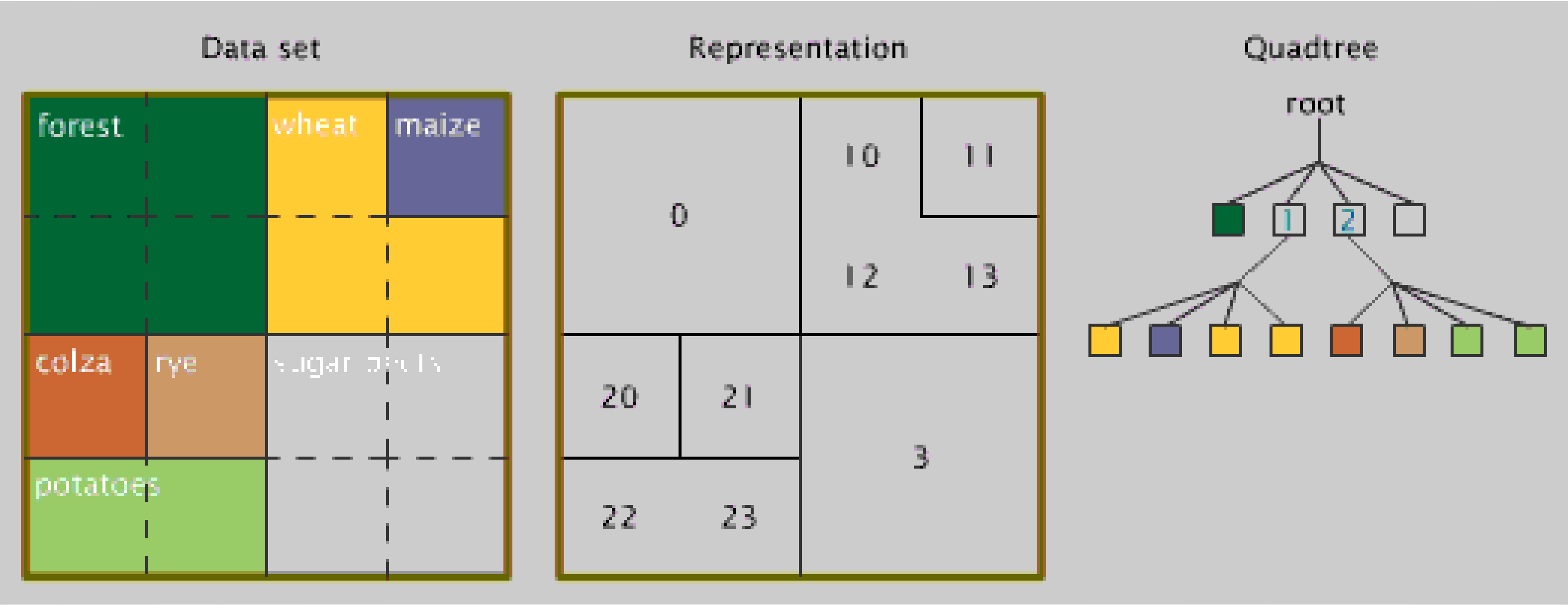
- In our case, a view is a rectangle that represents the subset of our world that we want to render at a particular time
- We are provided a class called `sf::View`
- With the view, we can scroll, zoom and rotate with ease
- Since our game's action occurs in a vertical corridor, we scroll the view at a constant speed towards the negative y
    - mView(0.f, -40 * dt.asSecond());
- sf::View::zoom(float factor) function to easily approach or move away from the center of the view
    - mView.zoom(0.2);
- sf::View::rotate(float degree) to add a rotation angle to the current one
- sf::View::setRotation(float degrees) to set the rotation of the view to an absolute value
    - mView.rotate(45);

View Optimization

- A Draw call is an expensive operation. We use culling to check if objects are within the viewing rectangle and then draw them.
- Game developers implement spatial subdivision: Dividing scene in multiple cells, which group all objects that reside within that given cell
- Cull a group of objects that are not in the view
- Quad Tree and Circle tree are two famous ways to subdivide space.

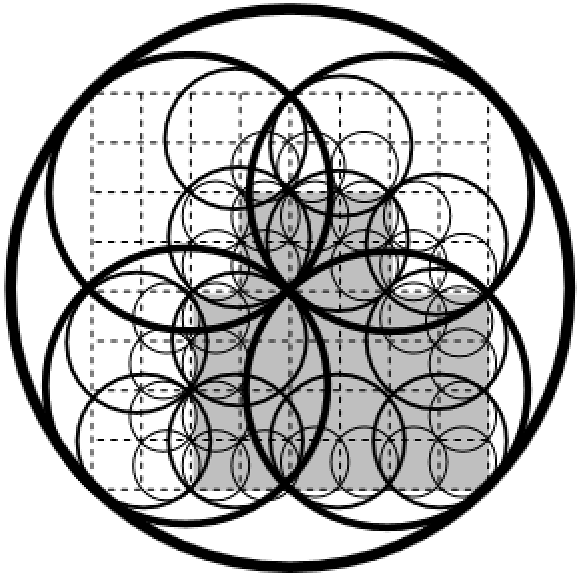
Quad Tree

- A quad tree is a hierarchical tree of cells. Only leaf nodes can contain objects and subdivide when a predetermined number of objects are present.



Circle Tree

- Similar to the quad tree, but instead each cell is a circle.
- Allows a different distribution of the objects



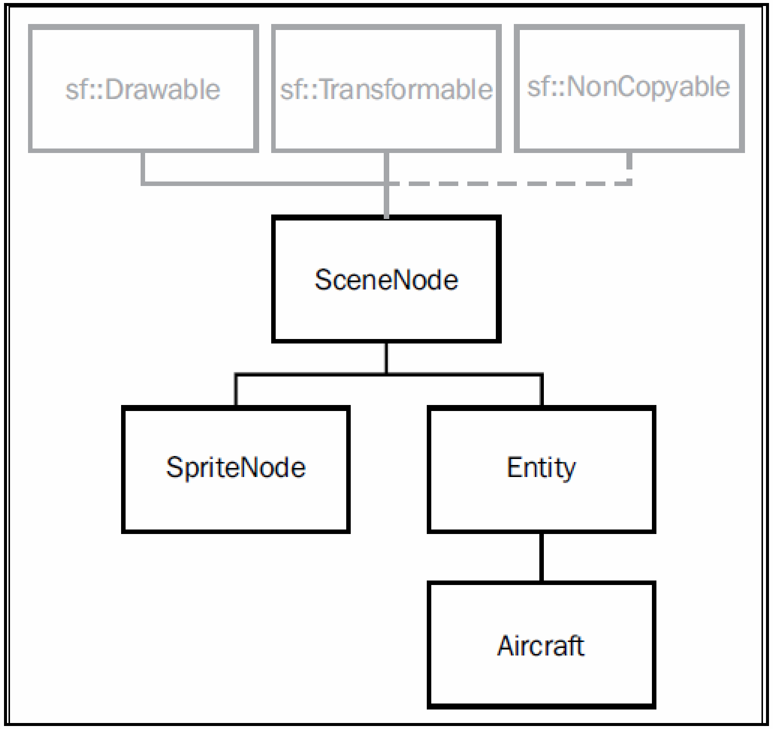
The SpriteNode Class

- The SpriteNode class represents a background sprite

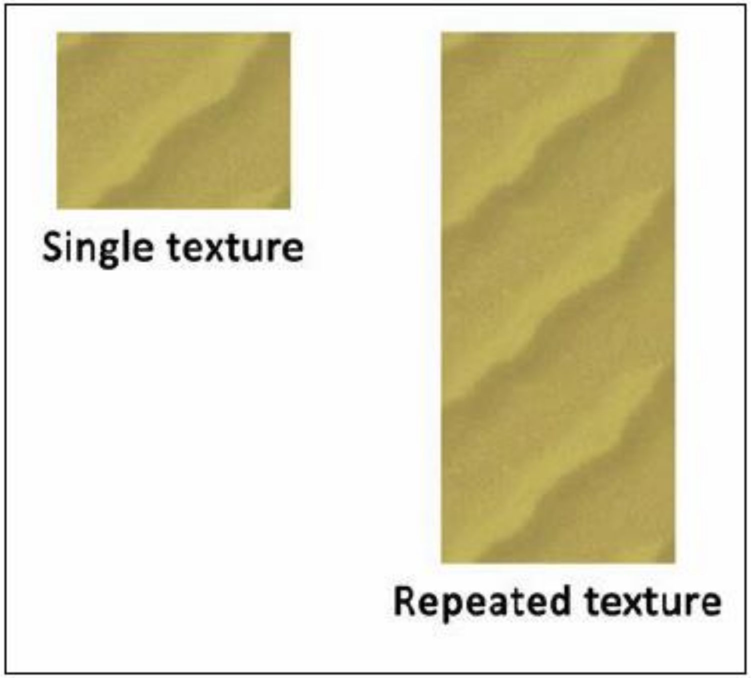
```
class SpriteNode : public SceneNode
{
public:
   explicit SpriteNode(const sf::Texture& texture);
   SpriteNode(const sf::Texture& texture, const sf::IntRect& rect);
private:
   virtual void drawCurrent(sf::RenderTarget& target,
                             sf::RenderStates states) const;
private:
   sf::Sprite mSprite;
};
```

The `SpriteNode` Class

- In the diagram below, the grey classes are part of SFML and the black ones are ours



Texture Repeating



- Every `sf::Texture` comes along with the option to enable repeating along both axis with the `sf::Texture::setRepeated(bool)` function

Composing the World

- The `World` class must contain all rendering data:
  - A reference to the render window
  - The world's current view
  - A texture holder with all the textures needed inside the world
  - The scene graph

  - Some pointers to access the scene graph's layer nodes
  - The bounding rectangle of the world, storing its dimensions
  - The position where the player's plane appears in the beginning
  - The speed with which the world is scrolled
  - A pointer to the player's aircraft

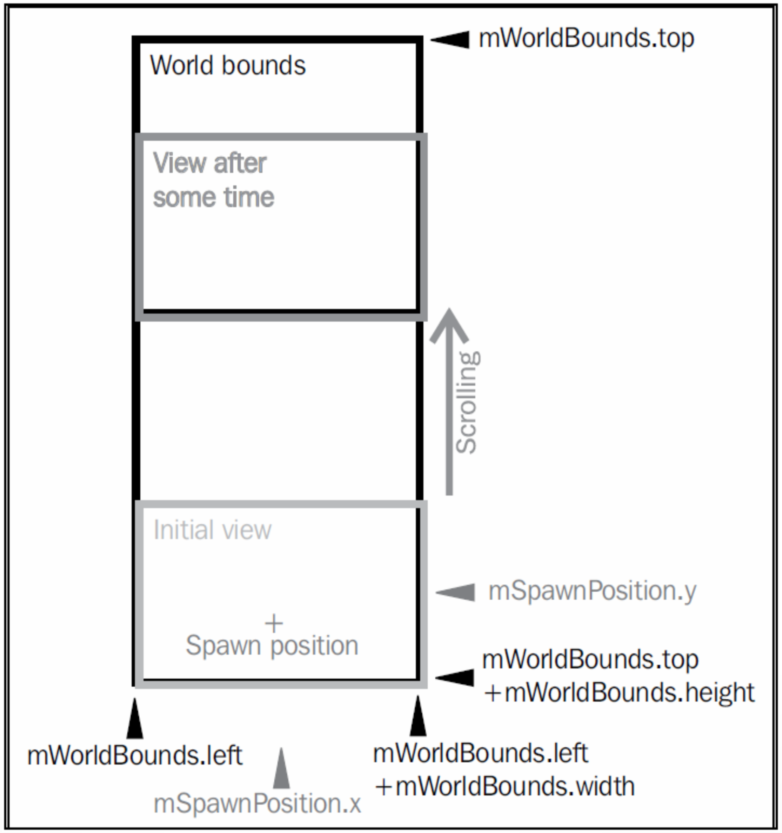
The World Class

```
class World : private
    sf::NonCopyable
{
public:
    explicit
    World(sf::RenderWindow&
    window);
    void update(sf::Time dt);
    void draw();
private:
    void loadTextures();
    void buildScene();
    private:
    enum Layer
    {
         Background,
         Air,
    LayerCount
    };
```

```
private:
    sf::RenderWindow& mWindow;
    sf::View mWorldView;
    TextureHolder mTextures;
    SceneNode mSceneGraph;
    std::array<SceneNode*, LayerCount>
    mSceneLayers;
    sf::FloatRect mWorldBounds;
    sf::Vector2f mSpawnPosition;
    float mScrollSpeed;
    Aircraft* mPlayerAircraft;
};
```

Composing the World

- The following diagram represents the world dimensions:



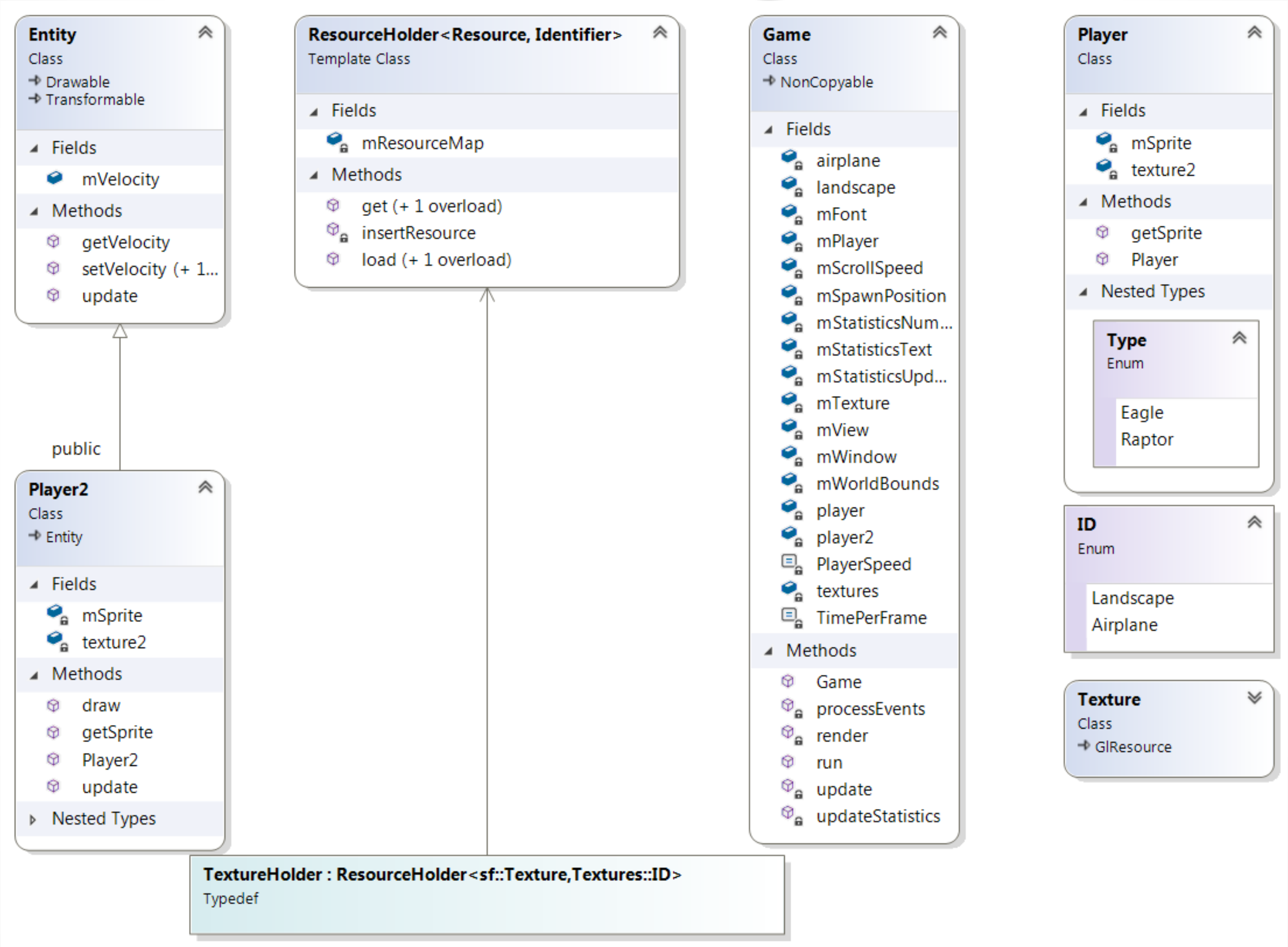
Loading the Textures

```
void World::loadTextures()
{
   mTextures.load(Textures::Eagle, "Media/Textures/Eagle.png");
   mTextures.load(Textures::Raptor, "Media/Textures/Raptor.png");
   mTextures.load(Textures::Desert, "Media/Textures/Desert.png");
}
```

What's Left?

- Remember that the main() function is our entry point
- We can start to look at everything from there, including all the classes we've looked at
- The scene is built in the `World::buildScene()` method
- The update() and draw() methods of World encapsulate scene graph functionality
- The run() function in main gets everything going

Entity



Entity
Class
Drawable
Transformable

Fields
mVelocity

Methods
getVelocity
setVelocity (+ 1...
update

public

Player2
Class
Entity

Fields
mSprite
texture2

Methods
draw
getSprite
Player2
update

Nested Types

ResourceHolder<Resource, Identifier>
Template Class

Fields
mResourceMap

Methods
get (+ 1 overload)
insertResource
load (+ 1 overload)

TextureHolder : ResourceHolder<sf::Texture,Textures::ID>
Typedef

Game
Class
NonCopyable

Fields
airplane
landscape
mFont
mPlayer
mScrollSpeed
mSpawnPosition
mStatisticsNum...
mStatisticsText
mStatisticsUpd...
mTexture
mView
mWindow
mWorldBounds
player
player2
PlayerSpeed
textures
TimePerFrame

Methods
Game
processEvents
render
run
update
updateStatistics

Player
Class

Fields
mSprite
texture2

Methods
getSprite
Player

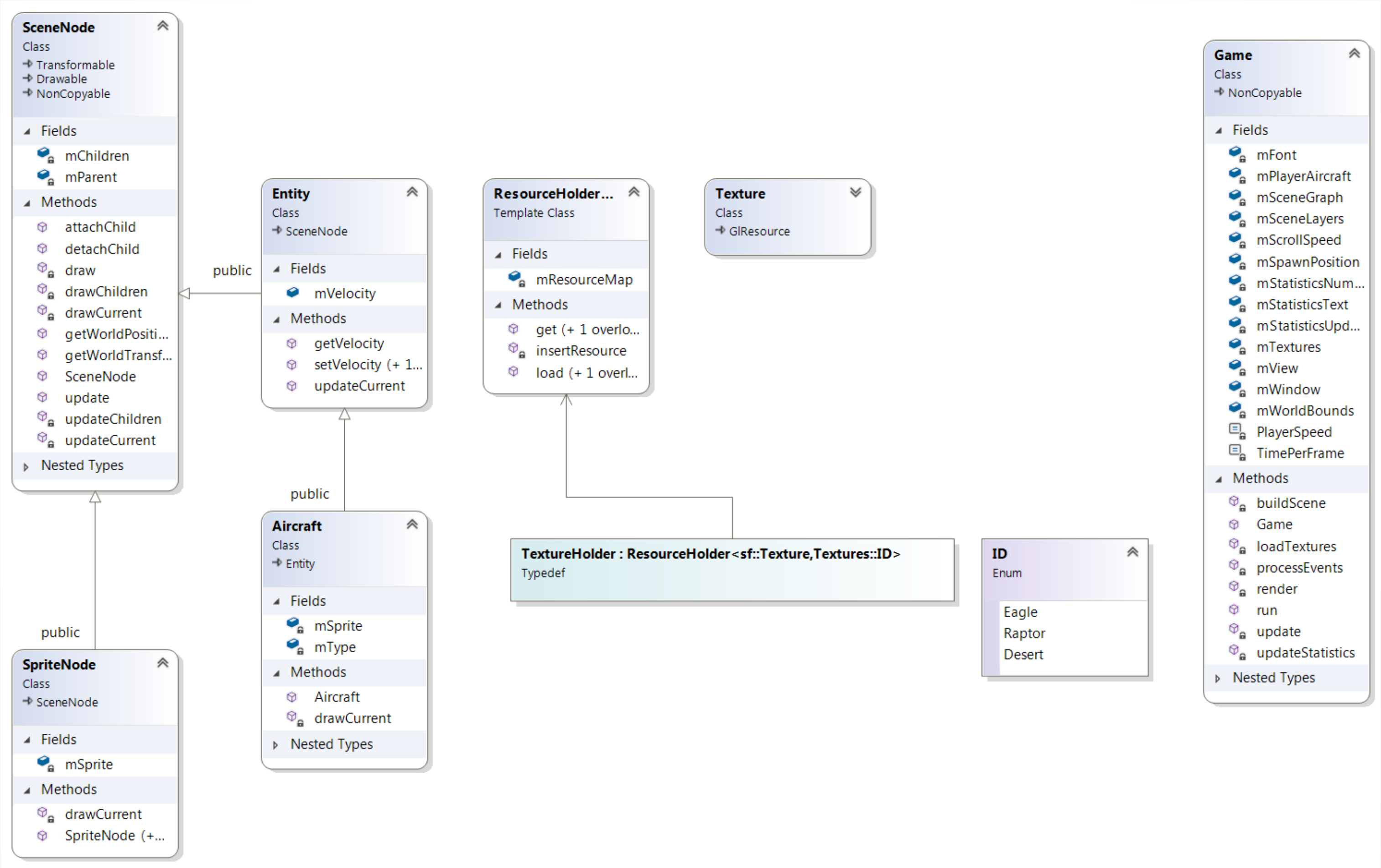
Nested Types

Type
Enum
Eagle
Raptor

ID
Enum
Landscape
Airplane

Texture
Class
GlResource

Scene Node



SceneNode
Class
→ Transformable
→ Drawable
→ NonCopyable

Fields
- mChildren
- mParent

Methods
- attachChild
- detachChild
- draw
- drawChildren
- drawCurrent
- getWorldPositi...
- getWorldTransf...
- SceneNode
- update
- updateChildren
- updateCurrent

Nested Types

public

Entity
Class
→ SceneNode

Fields
- mVelocity

Methods
- getVelocity
- setVelocity (+ 1...
- updateCurrent

public

Aircraft
Class
→ Entity

Fields
- mSprite
- mType

Methods
- Aircraft
- drawCurrent

Nested Types

public

SpriteNode
Class
→ SceneNode

Fields
- mSprite

Methods
- drawCurrent
- SpriteNode (+...

ResourceHolder...
Template Class

Fields
- mResourceMap

Methods
- get (+ 1 overlo...
- insertResource
- load (+ 1 overl...

Texture
Class
→ GlResource

TextureHolder : ResourceHolder<sf::Texture,Textures::ID>
Typedef

ID
Enum
- Eagle
- Raptor
- Desert

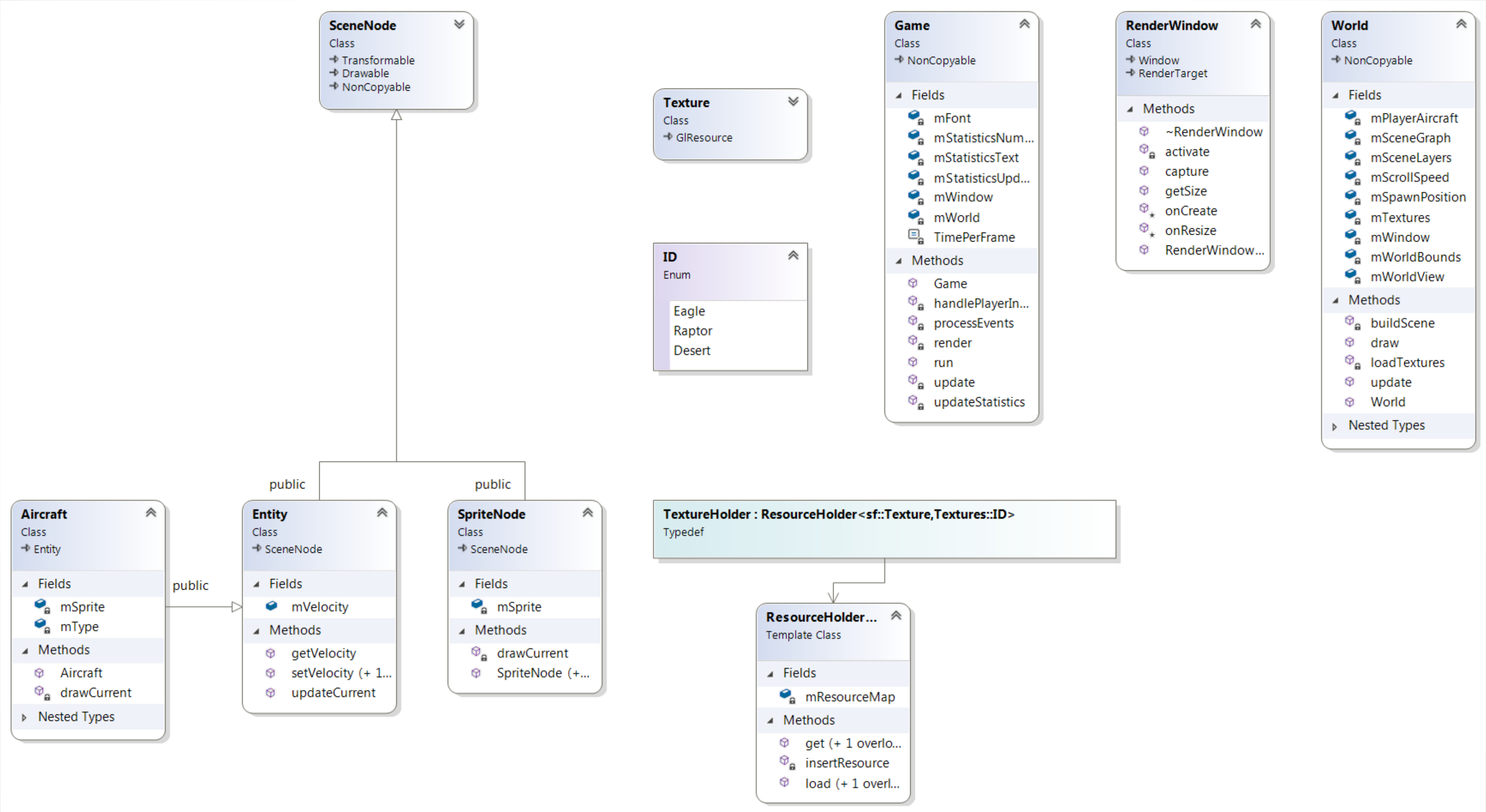
Game
Class
→ NonCopyable

Fields
- mFont
- mPlayerAircraft
- mSceneGraph
- mSceneLayers
- mScrollSpeed
- mSpawnPosition
- mStatisticsNum...
- mStatisticsText
- mStatisticsUpd...
- mTextures
- mView
- mWindow
- mWorldBounds
- PlayerSpeed
- TimePerFrame

Methods
- buildScene
- Game
- loadTextures
- processEvents
- render
- run
- update
- updateStatistics

Nested Types

World



SceneNode
Class
Transformable
Drawable
NonCopyable

Texture
Class
GlResource

ID
Enum
Eagle
Raptor
Desert

Game
Class
NonCopyable
Fields
mFont
mStatisticsNum...
mStatisticsText
mStatisticsUpd...
mWindow
mWorld
TimePerFrame
Methods
Game
handlePlayerIn...
processEvents
render
run
update
updateStatistics

RenderWindow
Class
Window
RenderTarget
Methods
~RenderWindow
activate
capture
getSize
onCreate
onResize
RenderWindow...

World
Class
NonCopyable
Fields
mPlayerAircraft
mSceneGraph
mSceneLayers
mScrollSpeed
mSpawnPosition
mTextures
mWindow
mWorldBounds
mWorldView
Methods
buildScene
draw
loadTextures
update
World
Nested Types

public
public
public

Aircraft
Class
Entity
Fields
mSprite
mType
Methods
Aircraft
drawCurrent
Nested Types

Entity
Class
SceneNode
Fields
mVelocity
Methods
getVelocity
setVelocity (+ 1...
updateCurrent

SpriteNode
Class
SceneNode
Fields
mSprite
Methods
drawCurrent
SpriteNode (+...

TextureHolder : ResourceHolder<sf::Texture,Textures::ID>
Typedef

ResourceHolder...
Template Class
Fields
mResourceMap
Methods
get (+ 1 overlo...
insertResource
load (+ 1 overl...