Game Engine Development II

Week2

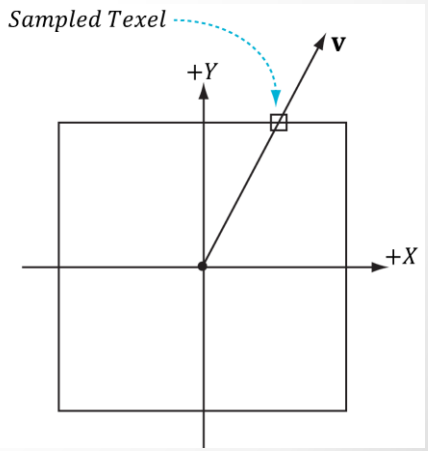
Hooman Salamat

Objectives

- 1. To learn what cube maps are and how to sample them in HLSL code.
- 2. To discover how to create cube maps with the DirectX texturing tools.
- 3. To find out how we can use cube maps to model reflections.
- 4. To understand how we can texture a sphere with cube maps to simulate a sky and distant mountains.

CUBE MAPPING

- The idea of cube mapping is to store six textures and to visualize them as the faces of a cube centered and axis aligned about some coordinate system.
- In Direct3D, a cube map is represented by a texture array with six elements such that:
- 1. index 0 refers to the +X face
- 2. index 1 refers to the –X face
- 3. index 2 refers to the +Y face
- 4. index 3 refers to the –Y face
- 5. index 4 refers to the +Z face
- 6. index 5 refers to the –Z face
- To identify a texel in a cube map, we use 3D texture coordinates, which define originating at the origin.
- We illustrate in 2D for simplicity; in 3D the square becomes a cube
- The square denotes the cube map centered and axis-aligned wit coordinate system.
- We shoot a vector **v** from the origin. The texel **v** intersects is the sa
- In this illustration, **v** intersects the cube face corresponding to the



TextureCube

In the HLSL, a cube texture is represented by the TextureCube type.

The lookup vector should be in the same space the cube map is relative to.

```
float3 v = float3(x, y, z); // some lookup vector
```

For example, if the cube map is relative to the world space (i.e., the cube faces are axis aligned with the world space axes), then the lookup vector should have world space coordinates.

```
// Common.hlsl
TextureCube gCubeMap : register(t0);
// Default.hlsl
VertexOut VS(VertexIn vin)
{
VertexOut vout;

// Use local vertex position as cubemap lookup vector.
vout.PosL = vin.PosL;

// Transform to world space.
float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);

// Always center sky about camera.
posW.xyz += gEyePosW;

// Set z = w so that z/w = 1 (i.e., skydome always on far plane).
vout.PosH = mul(posW, gViewProj).xyww;

return vout;
}


float4 PS(VertexOut pin) : SV_Target
{
return gCubeMap.Sample(gsamLinearWrap, pin.PosL);
}
```

ENVIRONMENT MAPS

The primary application of cube maps is *environment mapping.*

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 point the camera to 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.

The field of view angle 90° will capture the entire surrounding environment.

An environment map is a cube map where the cube faces store the surrounding images of an environment.

Environment Map

For simplicity, we omit certain objects from the scene. For example, the environment map in this figure only captures the distant "background" information of the sky and mountains that are very far away.

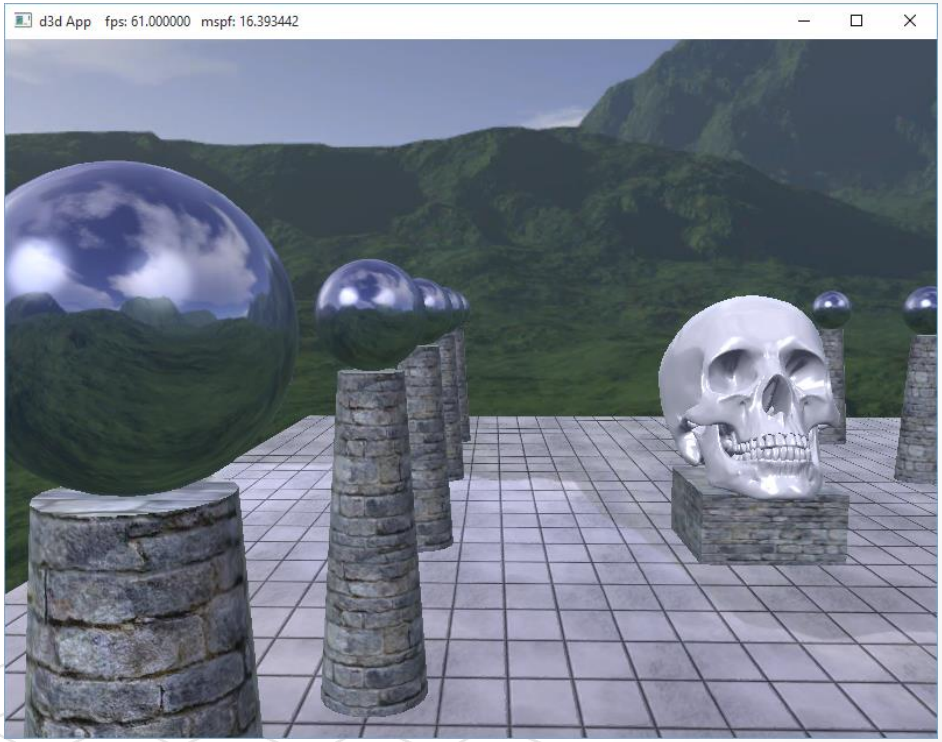
For outdoor environment maps, the program *Terragen:*

http://www.planetside.co.uk/

Terragen is common to use (free for personal use), and can create photorealistic outdoor scenes.

The environment maps in our demos were made with *Terragen. The following link gives you a* Terragen *script that you can use as a sample*.

https://developer.valvesoftware.com/wiki/Skybox_(2D)_with_Terragen uses the current camera position, and render out the six surrounding images with a 90° field of view.



texassemble

Once you have created the six cube map images using some program, we need to create a cube map texture, which stores all six.

The DDS texture image format we have been using readily supports cube maps, and we can use the *texassemble* tool to build a cube map from six images.

This is an example of how to create a cube map using texassemble:

NVIDIA provides Photoshop plugins for saving .DDS and cubemaps in Photoshop; see https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop

texassemble -cube -w 256 -h 256 -o cubemap.dds lobbyxpos.jpg lobbyxneg.jpg lobbyypos.jpg lobbyyneg.jpg lobbyzpos.jpg lobbyzneg.jpg

Loading and Using Cube Maps in Direct3D

DDS texture loading code (*DDSTextureLoader.h/.cpp*) already supports loading cube maps, and we can load the texture like any other.

The loading code will detect that the DDS file contains a cube map, and will create a texture array and load the face data into each element.

```
void CubeMapApp::LoadTextures()
{
    std::vector<std::string> texNames =
    {
        "bricksDiffuseMap",
        "tileDiffuseMap",
        "defaultDiffuseMap",
        "skyCubeMap"
    };

    std::vector<std::wstring> texFilenames =
    {
        L"../../Textures/bricks2.dds",
        L"../../Textures/tile.dds",
        L"../../Textures/white1x1.dds",
        L"../../Textures/grasscube1024.dds"
    };

    for (int i = 0; i < (int)texNames.size(); ++i)
    {
        auto texMap = std::make_unique<Texture>();
        texMap->Name = texNames[i];
        texMap->Filename = texFilenames[i];
        ThrowIfFailed(DirectX::CreateDDSTextureFromFile12(md3dDevice.Get(),
            mCommandList.Get(), texMap->Filename.c_str(),
            texMap->Resource, texMap->UploadHeap));

        mTextures[texMap->Name] = std::move(texMap);
    }
}
```

CubeMapApp::BuildDescriptorHeaps

```
void CubeMapApp::BuildDescriptorHeaps()
{
//
// Create the SRV heap.
//
D3D12_DESCRIPTOR_HEAP_DESC srvHeapDesc = {};
srvHeapDesc.NumDescriptors = 4;
srvHeapDesc.Type = D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV;
srvHeapDesc.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE;
ThrowIfFailed(md3dDevice->CreateDescriptorHeap(&srvHeapDesc, IID_PPV_ARGS(&mSrvDescriptorHeap)));

// Fill out the heap with actual descriptors.
//
CD3DX12_CPU_DESCRIPTOR_HANDLE hDescriptor(mSrvDescriptorHeap->GetCPUDescriptorHandleForHeapStart());

auto bricksTex = mTextures["bricksDiffuseMap"]->Resource;
auto tileTex = mTextures["tileDiffuseMap"]->Resource;
auto whiteTex = mTextures["defaultDiffuseMap"]->Resource;
auto skyTex = mTextures["skyCubeMap"]->Resource;

D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = bricksTex->GetDesc().Format;
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.MipLevels = bricksTex->GetDesc().MipLevels;
srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;
md3dDevice->CreateShaderResourceView(bricksTex.Get(), &srvDesc, hDescriptor);
```

Creating an SRV to a cube map texture resource

When we create an SRV to a cube map texture resource, we specify the dimension D3D12_SRV_DIMENSION_TEXTURECUBE and use the TextureCube property of the SRV description:

```
// next descriptor
hDescriptor.Offset(1, mCbvSrvDescriptorSize);

srvDesc.Format = tileTex->GetDesc().Format;
srvDesc.Texture2D.MipLevels = tileTex->GetDesc().MipLevels;
md3dDevice->CreateShaderResourceView(tileTex.Get(), &srvDesc, hDescriptor);

// next descriptor
hDescriptor.Offset(1, mCbvSrvDescriptorSize);

srvDesc.Format = whiteTex->GetDesc().Format;
srvDesc.Texture2D.MipLevels = whiteTex->GetDesc().MipLevels;
md3dDevice->CreateShaderResourceView(whiteTex.Get(), &srvDesc, hDescriptor);

// next descriptor
hDescriptor.Offset(1, mCbvSrvDescriptorSize);

srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURECUBE;
srvDesc.TextureCube.MostDetailedMip = 0;
srvDesc.TextureCube.MipLevels = skyTex->GetDesc().MipLevels;
srvDesc.TextureCube.ResourceMinLODClamp = 0.0f;
srvDesc.Format = skyTex->GetDesc().Format;
md3dDevice->CreateShaderResourceView(skyTex.Get(), &srvDesc, hDescriptor);

mSkyTexHeapIndex = 3;
}
```

TEXTURING A SKY

We create a large sphere that surrounds the entire scene.

To create the illusion of distant mountains far in the horizon and a sky, we texture the sphere using an environment map by the method.

We illustrate in 2D for simplicity; in 3D the square becomes a cube and the circle becomes a sphere.

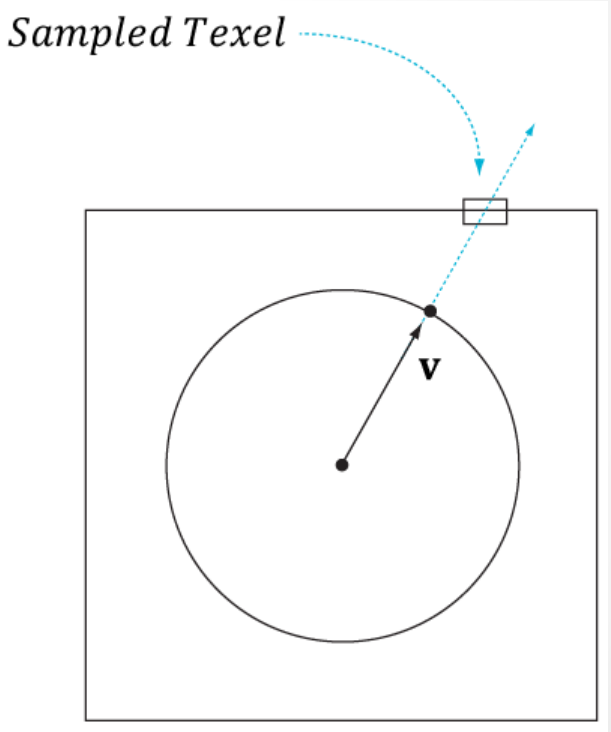
We assume that the sky and environment map are centered about the same origin.

To texture a point on the surface of the sphere, we use the vector from the origin to the surface point as the lookup vector into the cube map. This projects the cube map onto the sphere's surface.

We assume that the sky sphere is infinitely far away (i.e., it is centered about the world space but has infinite radius), and so no matter how the camera moves in the world, we never appear to get closer or farther from the surface of the sky sphere.

To implement this infinitely faraway sky, we simply center the sky sphere about the camera in world space so that it is always centered about the camera.

As the camera moves, we are getting no closer to the surface of the sphere.



The shader file for the sky

The sky shader programs are significantly different than the shader programs for drawing our objects (*Default.hlsl*).

However, it shares the same root signature so that we do not have to change root signatures in the middle of drawing.

The code that is common to both *Default.hlsl* and *Sky.hlsl* has been moved to *Common.hlsl* so that the code is not duplicated.

```
#include "Common.hlsl"

struct VertexIn
{
float3 PosL    : POSITION;
float3 NormalL : NORMAL;
float2 TexC    : TEXCOORD;
};

struct VertexOut
{
float4 PosH : SV_POSITION;
    float3 PosL : POSITION;
};

VertexOut VS(VertexIn vin)
{
VertexOut vout;

// Use local vertex position as cubemap lookup vector.
vout.PosL = vin.PosL;

// Transform to world space.
float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);

// Always center sky about camera.
posW.xyz += gEyePosW;

// Set z = w so that z/w = 1 (i.e., skydome always on far plane).
vout.PosH = mul(posW, gViewProj).xyww;

return vout;
}

float4 PS(VertexOut pin) : SV_Target
{
return gCubeMap.Sample(gsamLinearWrap, pin.PosL);
}
```

CubeMapApp::BuildRootSignature

```
void CubeMapApp::BuildRootSignature()
{
CD3DX12_DESCRIPTOR_RANGE texTable0;
texTable0.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 1, 0, 0); //one SRV descriptor in space0

CD3DX12_DESCRIPTOR_RANGE texTable1;
texTable1.Init(D3D12_DESCRIPTOR_RANGE_TYPE_SRV, 4, 1, 0); //4 SRV descriptor in space1

// Root parameter can be a table, root descriptor or root constants.
CD3DX12_ROOT_PARAMETER slotRootParameter[5];

// Perfomance TIP: Order from most frequent to least frequent.
slotRootParameter[0].InitAsConstantBufferView(0); //b0,space0
slotRootParameter[1].InitAsConstantBufferView(1); //b1, space0
slotRootParameter[2].InitAsShaderResourceView(0, 1); //t0, space1
slotRootParameter[3].InitAsDescriptorTable(1, &texTable0, D3D12_SHADER_VISIBILITY_PIXEL); //t0 in space0
slotRootParameter[4].InitAsDescriptorTable(1, &texTable1, D3D12_SHADER_VISIBILITY_PIXEL); //t1,t2,t3,t4 in space1


auto staticSamplers = GetStaticSamplers();

// A root signature is an array of root parameters.
CD3DX12_ROOT_SIGNATURE_DESC rootSigDesc(5, slotRootParameter,
(UINT)staticSamplers.size(), staticSamplers.data(),
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
```

SetPipelineState

Drawing the sky requires different shader programs, and therefore a new sky PSO.

We draw the sky as a separate layer in our drawing code.

First, we bind the sky cube map. For our demos, we just use one "world" cube map representing the environment from far away, so all objects will use the same cube map and we only need to set it once per-frame.

If we wanted to use "local" cube maps, we would have to change them per-object, or dynamically index into an array of cube maps.

Then we bind all the textures used in this scene. Note that we only have to specify the first descriptor in the table.

The root signature knows how many descriptors are expected in the table.

```
CD3DX12_GPU_DESCRIPTOR_HANDLE skyTexDescriptor(mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());

skyTexDescriptor.Offset(mSkyTexHeapIndex, mCbvSrvDescriptorSize);

mCommandList->SetGraphicsRootDescriptorTable(3, skyTexDescriptor);

mCommandList->SetGraphicsRootDescriptorTable(4, mSrvDescriptorHeap->GetGPUDescriptorHandleForHeapStart());

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));
```

PSO for sky

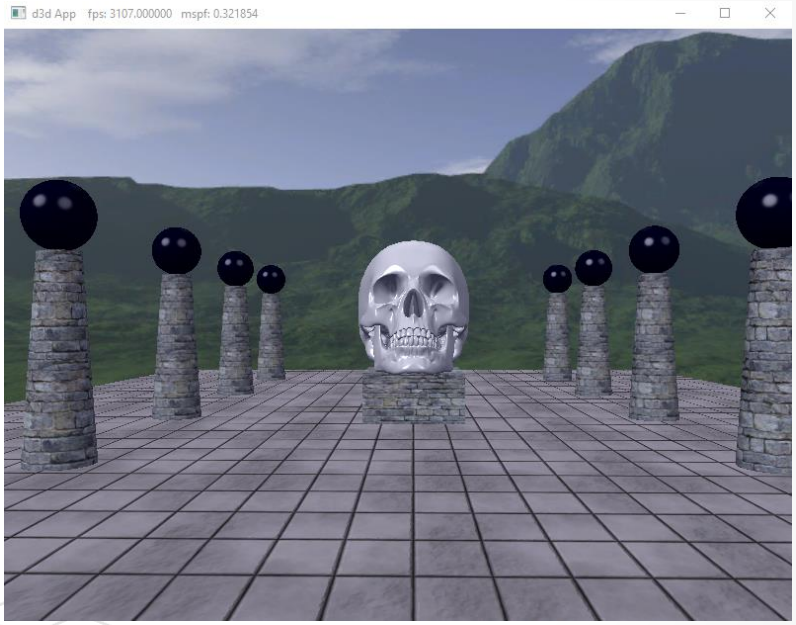
In addition, rendering the sky requires some different render states.

In particular, because the camera lies inside the sphere, we need to disable back face culling (or making counterclockwise triangles front facing would also work), and we need to change the depth comparison function to LESS_EQUAL so that the sky will pass the depth test:

```
void CubeMapApp::BuildPSOs()
{
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;
// PSO for opaque objects.
…….

D3D12_GRAPHICS_PIPELINE_STATE_DESC skyPsoDesc = opaquePsoDesc;

// The camera is inside the sky sphere, so just turn off culling.
skyPsoDesc.RasterizerState.CullMode = D3D12_CULL_MODE_NONE;

// Make sure the depth function is LESS_EQUAL and not just LESS.
// Otherwise, the normalized depth values at z = 1 (NDC) will
// fail the depth test if the depth buffer was cleared to 1.
skyPsoDesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_LESS_EQUAL;
skyPsoDesc.pRootSignature = mRootSignature.Get();
skyPsoDesc.VS =
{
reinterpret_cast<BYTE*>(mShaders["skyVS"]->GetBufferPointer()),
mShaders["skyVS"]->GetBufferSize()
};
skyPsoDesc.PS =
{
reinterpret_cast<BYTE*>(mShaders["skyPS"]->GetBufferPointer()),
mShaders["skyPS"]->GetBufferSize()
};
ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&skyPsoDesc, IID_PPV_ARGS(&mPSOs["sky"])));
```

CubeMap Demo

```
auto mirror0 = std::make_unique<Material>();

mirror0->Name = "mirror0";

mirror0->MatCBIndex = 2;

mirror0->DiffuseSrvHeapIndex = 2;

mirror0->DiffuseAlbedo = XMFLOAT4(0.0f, 0.0f, 0.1f, 1.0f);

mirror0->FresnelR0 = XMFLOAT3(0.98f, 0.97f, 0.95f);

mirror0->Roughness = 0.1f;
```



CubeMapApp::BuildRenderItems

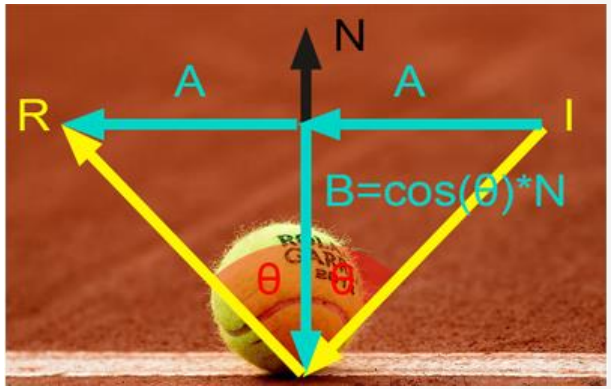
```
XMStoreFloat4x4(&rightSphereRitem->World, rightSphereWorld);

rightSphereRitem->TexTransform = MathHelper::Identity4x4();

rightSphereRitem->ObjCBIndex = objCBIndex++;

rightSphereRitem->Mat = mMaterials["mirror0"].get();

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;
```

Reflect function in hlsl

Returns a reflection vector using an incident ray and a surface normal.

I=A+B,R=A−B.

The vector B can easily be computed. It is the projection of the vector I or R onto the vector N



B=cos(θ)∗N

I=A+cos(θ)∗N,R=A−cos(θ)∗N

A=I−cos(θ)∗N

R=I−cos(θ)∗N−cos(θ)∗N

R= *I*−2cos(θ)*N*

R= *I*−2(*N*·*I*)*N*

***ret* reflect(*i*, *n*)** function calculates the reflection vector using the following formula: $r = i - 2 * n * dot(i, n)$

MODELING REFLECTIONS

How to use environment maps to model *specular reflections* coming from the surrounding environment?

By specular reflections, we mean that we are just going to look at the light that is reflected off a surface due to the Fresnel effect.

When we render a scene about a point *P* to build an environment map, the environment map stores the light values coming in from every direction about the point *P*.

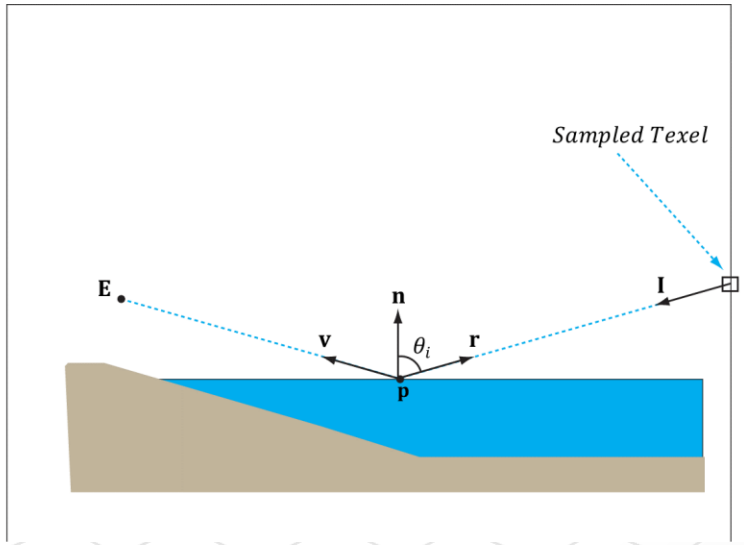
We use this data to approximate specular reflections of light coming from the surrounding environment.

We can think of every texel on the environment map as a source of light.

Light from the environment comes in with incident direction I and reflects off the surface (due to the Fresnel effect) and enters the eye in the direction.

Here **E** is the eye point, and **n** is the surface normal at the point **p**. The texel that stores the light that reflects off **p** and enters the eye is obtained by sampling the cube map with the vector **r**.

v = **E** − **p** => **r** = reflect(−**v**, **n**)



Compute the reflection vector per-pixel

```
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(gsamAnisotropic
Wrap, 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;

// Add in specular reflections.

float3 r = reflect(-toEyeW, pin.NormalW);
float4 reflectionColor = gCubeMap.Sample(gsamLinearWrap, r);
```

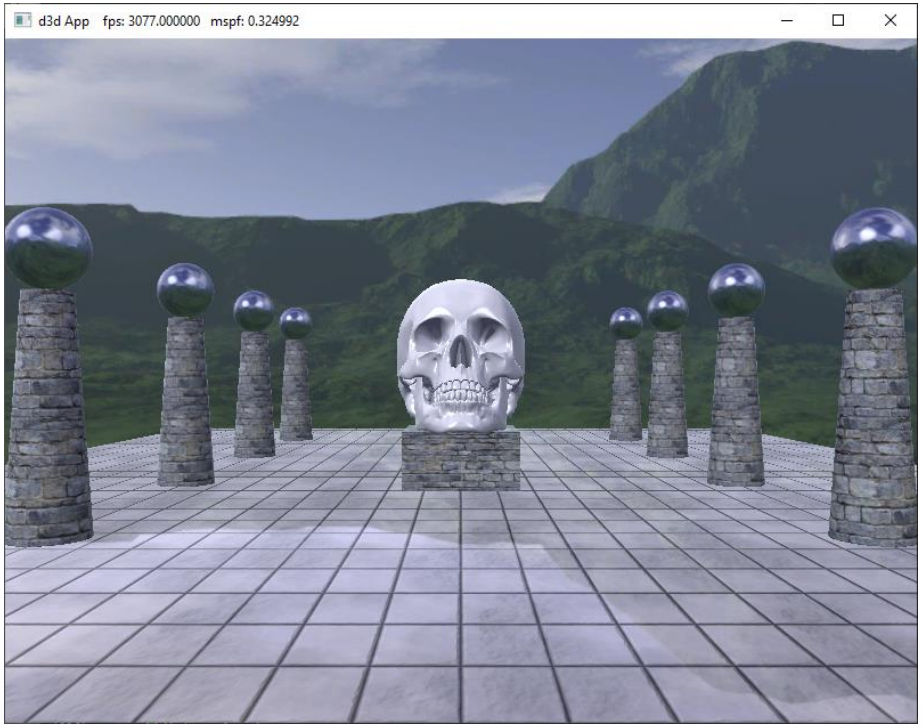
//the Fresnel effect determines how much light is reflected from the environment into the eye based on the material properties of the surface and the angle between the light vector (reflection vector) and normal.

```
float3 fresnelFactor = SchlickFresnel(fresnelR0, pin.NormalW, r);
```

//In addition, we scale the amount of reflection based on the shininess of the material—a rough material should have a low amount of reflection, but still some reflection.

```
litColor.rgb += shininess * fresnelFactor * reflectionColor.rgb;

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

CubeMap Demo



Flat Surfaces

The reflections via environment mapping do not work well for flat surfaces.

This is because the reflection vector does not tell the whole story, as it does not incorporate position.

The reflection vector corresponding to two different points **p** and **p**′ when the eye is at positions **E** and **E**′, respectively.

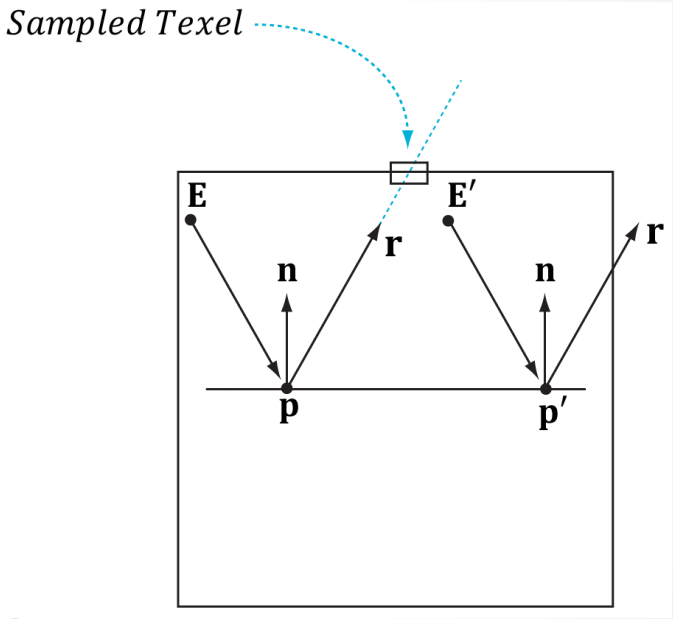
A ray has position and direction, whereas a vector just has direction

We really need a reflection ray and to intersect the ray with the environment map.

From the figure, we see that the two reflection rays, $\mathbf{q}(t) = \mathbf{p} + t\mathbf{r}$ and $\mathbf{q}'(t) = \mathbf{p}' + t\mathbf{r}$, intersect different texels of the cube map, and thus should be colored differently.

However, because both rays have the same direction vector **r**, and the direction vector **r** is solely used for the cube map lookup, the same texel gets mapped to **p** and **p**′ when the eye is at **E** and **E**′, respectively.

For curvy surfaces, this shortcoming of environment mapping goes largely unnoticed, since the curvature of the surface causes the reflection vector to vary.



Cube Map Lookup

One solution is to associate some proxy geometry with the environment map.

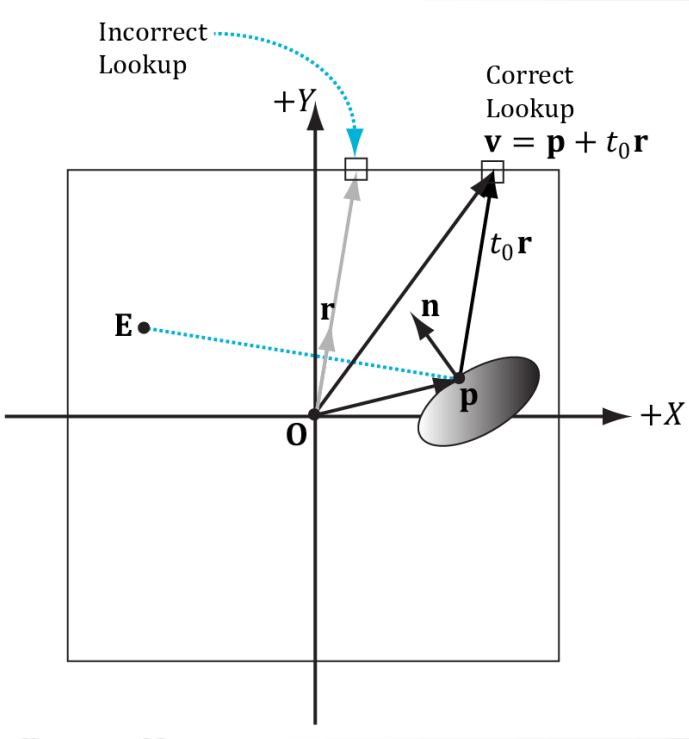
For example, suppose we have an environment map for a square room. We can associate an axis-aligned bounding box with the environment map that has approximately the same dimensions as the room.

Figure shows how we can do a ray intersection with the box to compute the vector **v** which gives a better lookup vector than the reflection vector **r**.

In stead of using the reflection vector **r** for the cube map lookup, we use the intersection point $\mathbf{v} = \mathbf{p} + t_0\mathbf{r}$ between the ray and the box.

Note that the point **p** is made relative to the center of the bounding box proxy geometry so that the intersection point can be used as a lookup vector for the cube map.

If the bounding box associated with the cube map gets sent to the shader (e.g., via a constant buffer), then the ray/box intersection test can be done in the pixel shader, and we can compute the improved lookup vector in the pixel shader to sample the cube map.



Cube Map Lookup

The following function shows how the cube map look up vector can be computed.

```
float3 BoxCubeMapLookup(float3 rayOrigin, float3
unitRayDir, float3 boxCenter, float3 boxExtents)
{
// Based on slab method as described in Real-Time Rendering
// Make relative to the box center.
float3 p = rayOrigin - boxCenter;
// The ith slab ray/plane intersection formulas for AABB are :
// t1 = (-dot(n_i, p) + h_i)/dot(n_i, d) = (-p_i + h_i) / d_i
// t2 = (-dot(n_i, p) - h_i)/dot(n_i, d) = (-p_i - h_i) / d_i
// Vectorize and do ray/plane formulas for every slab together.
float3 t1 = (-p + boxExtents) / unitRayDir;
float3 t2 = (-p - boxExtents) / unitRayDir;
// Find max for each coordinate. Because we assume the ray is inside
// the box, we only want the max intersection parameter.
float3 tmax = max(t1, t2);
// Take minimum of all the tmax components:
float t = min(min(tmax.x, tmax.y), tmax.z);
// This is relative to the box center so it can be used as a
// cube map lookup vector.
return p + t * unitRayDir;
}
```

Game Loop Programming in SFML

Objectives

- Explore an example game loop
- Review and apply the concepts of frames, frame rates and fixed time steps
- Create and display sprites to the game window

How to set an Icon for the window

- sf::Image mIcon;
- mIcon.loadFromFile("Media/Textures/icon.png");
- mWindow.setIcon(mIcon.getSize().x, mIcon.getSize().y, mIcon.getPixelsPtr());
- sf::Image also provides functions to load, read, write pixels
- mIcon.create(20, 20, sf::Color::Yellow);
- sf::Color color = mIcon.getPixel(0, 0);
- color.a = 0; //make the top-left pixel transparent
- color.r = 0; //set the r = 0 (rgb) from the color
- mIcon.setPixel(0, 0, color);

Font and Text

- Create a graphical text to display

```
sf::Font mFont;
sf::Text mText;
if (!mFont.loadFromFile("Media/Sansation.ttf"))
return;


mText.setString("Hello SFML");
mText.setFont(mFont);
mText.setPosition(5.f, 5.f);
mText.setCharacterSize(50);
mText.setFillColor(sf::Color::Black);
```

Play the Music

- Streamed music played from an audio file
- Musics are sounds that are streamed rather than completely loaded in memory
- A music is played in its own thread in order not block the rest of the program
- Supported audio formats: ogg, wav, flac, aiff, au, raw, paf, svx, nist, voc, ircam, w64, mat4, mat5, pvf, htk, sds, avr, sd2, caf, wve, mpc2k, rf64

Music Parameters

- #include <SFML/Audio.hpp>
- sf::Music mMusic;
- mMusic.openFromFile("Media/Textures/nice_music.ogg");
- //change some parameters
- mMusic.setPosition(0, 1, 10); //change its 3D position
- mMusic.setPitch(2); //increase the pitch
- mMusic.setVolume(50); //reduce the volume
- mMusic.setLoop(true); //make it loop
- mMusic.setAttenuation(100);
- mMusic.play();

Game Loop

- The run() function you saw in the example from last week, and below, is known as the main loop or game loop

```
void Game::run()
{
   while (mWindow.isOpen())
   {
       processEvents();
       update();
       render();
   }
}
```

Game Loop (cont'd.)

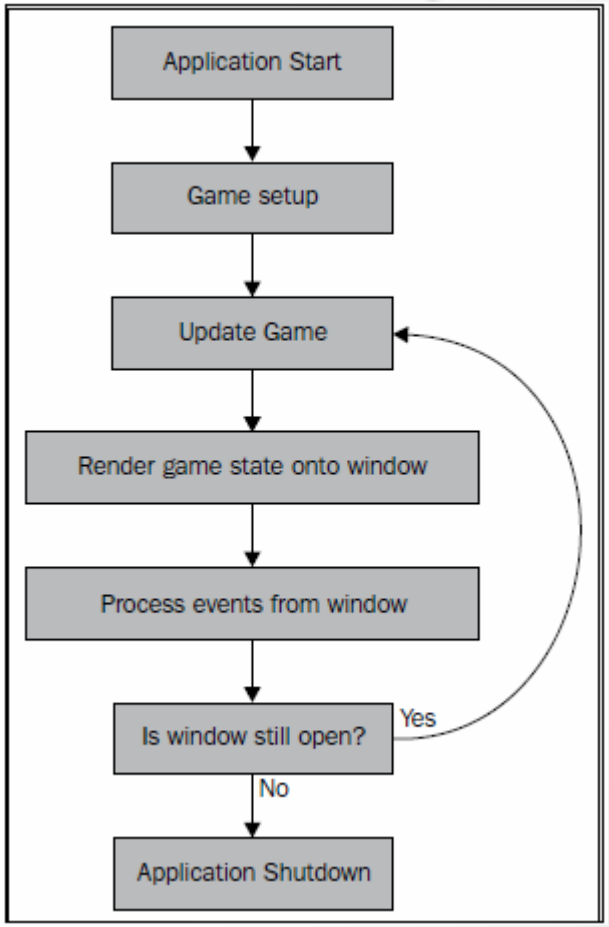
- It processes all the components in the game and continues to do so until the application is terminated
- The processing of events, updating of all game assets and then rendering them to the destination output is a standard loop for games

Game Loop (cont'd.)

- It processes all the components in the game and continues to do so until the application is terminated
- The processing of events, updating of all game assets and then rendering them to the destination output is a standard loop for games
- You've heard the term **frame** or **tick** before, and that is what we call an iteration of the loop

Game Loop (cont'd.)

- The flowchart to the right illustrates the logic and different processes of the game, including the main loop



Events

- Events can be user-generated such as mouse clicks or movement or keyboard presses
- They can also be generated by the assets in the game, such as when an enemy spots the player
- We don't have any events in our example yet, so let's create some!

- How 'bout moving that circle with the keyboard?

Events (cont'd.)

- For events in SFML, we use the `sf::Event` object
- We're going to use two events for this example:

  `sf::Event::KeyPressed` and `sf::Event::KeyReleased`

processEvents()

```
void Game::processEvents()
{
    sf::Event event;
    while (mWindow.pollEvent(event))
    {
        switch (event.type)
        {
            case sf::Event::KeyPressed:
                handlePlayerInput(event.key.code, true);
                break;
            case sf::Event::KeyReleased:
                handlePlayerInput(event.key.code, false);
                break;
            case sf::Event::Closed:
                mWindow.close();
                break;
        }
    }
}
```

handlePlayerInput()

```
void Game::handlePlayerInput(sf::Keyboard::Key key, bool isPressed)
{
        if (key == sf::Keyboard::W)
          mIsMovingUp = isPressed;
        else if (key == sf::Keyboard::S)
          mIsMovingDown = isPressed;
        else if (key == sf::Keyboard::A)
          mIsMovingLeft = isPressed;
        else if (key == sf::Keyboard::D)
          mIsMovingRight = isPressed;
}
```

New update()

```
void Game::update()
{
    sf::Vector2f movement(0.f, 0.f);
    if (mIsMovingUp)
      movement.y -= 1.f;
    if (mIsMovingDown)
      movement.y += 1.f;
    if (mIsMovingLeft)
      movement.x -= 1.f;
    if (mIsMovingRight)
      movement.x += 1.f;

    mPlayer.move(movement);
}
```

How about mouse?

```
if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {
sf::Vector2i mousePosition =
sf::Mouse::getPosition(mWindow);
mPlayer.setPosition((float)mousePosition.x,
(float)mousePosition.y);
}
```

Vector Object

- SFML's Vector object is instantiated as:

```
sf::Vector2<float>
```

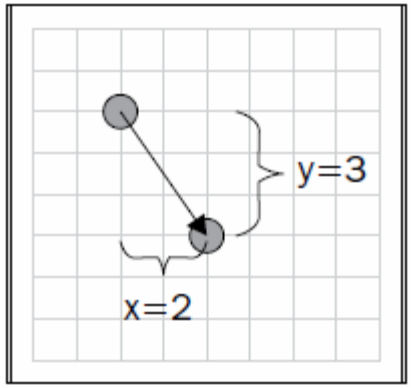
- We use the typedef for variable declarations, which is as follows:

```
sf::Vector2f myVector(0.f, 0.f);
```

- As you would expect, there are many common operations in the Vector class that we can access through a Vector object

Vector Object (cont'd.)

- In games, vectors can represent coordinates or a direction to move
- The diagram below represents a vector(2,3) and it could be a translation of 2 units to the right and 3 down:



Frame-Independence

- You might remember from Unity that we were able to move an object a certain number of units per second
  - We multiplied the speed by Time.deltaTime

- We can do this in SFML too, so that the player's movement isn't dependent on the framerate – or number of times the update runs per second

Frame-Independence (cont'd.)

- Well, we're still relying on the framerate, but the movement is spread out evenly over the frames

- So let's have a look at our new update function and see what's new...

New update()

```
void Game::update(sf::Time deltaTime)
{
    sf::Vector2f movement(0.f, 0.f);
    if (mIsMovingUp)
      movement.y -= PlayerSpeed;
    if (mIsMovingDown)
      movement.y += PlayerSpeed;
    if (mIsMovingLeft)
      movement.x -= PlayerSpeed;
    if (mIsMovingRight)
      movement.x += PlayerSpeed;

    mPlayer.move(movement * deltaTime.asSeconds());
}
```

Measuring Frames

- We can measure the time each frame takes in order to figure out `deltaTime`
- We use the `sf::Clock` class
- Sf::Clock has only two methods: getElapsedTime() and restart(). Both returns the elapsed time since the clock was started and then it resets the clock to zero. getElapsedTime() can be called without calling restart()

```
sf::Clock clock;
sf::Time time = clock.getElapsedTime();
float seconds = time.asSeconds();
sf::Int32 milliseconds = time.asMilliseconds();
sf::Int64 microseconds = time.asMicroseconds();
time = clock.restart();
```

Measuring Frames (cont'd.)

```
void Game::run()
{
    sf::Clock clock;
    while (mWindow.isOpen())
    {
        sf::Time deltaTime = clock.restart();
        processEvents();
        update(deltaTime);
        render();
    }
}
```

Fixed Time Step

- Time based on a system function such as a while loop will never be constant
  - We saw this way back with HTML5 and its highly-varying frame rate
  - Unity too!
- Fortunately we can create fixed time execution using a counter and a check
  - The while loop is definitely going to execute fast enough to serve as very small time increments added to our counter variable

Fixed Time Step (cont'd.)

```
void Game::run()
{
    sf::Clock clock;
    sf::Time timeSinceLastUpdate = sf::Time::Zero;
    while (mWindow.isOpen())
    {
        processEvents();
        timeSinceLastUpdate += clock.restart();
        while (timeSinceLastUpdate > TimePerFrame)
        {
            timeSinceLastUpdate -= TimePerFrame;
            processEvents();
            update(TimePerFrame);
        }
        render();
    }
}
```

Fixed Time Step (cont'd.)

- If you want to read more on this topic, you can read the article at the following address:
  - http://gafferongames.com/game-physics/fix-your-timestep

Displaying Sprites

```
sf::Texture texture;
if (!texture.loadFromFile("path/to/file.png"))
{
  // Handle loading error
}
sf::Sprite sprite(texture);
sprite.setPosition(100.f, 100.f);
window.clear();
window.draw(sprite);
window.display();
```

Rendering

- Rendering is the process of drawing your assets to the screen
- Ideally, we'd only want our assets being drawn if they were updated somehow in the program
  - Would save on performance
- However, *real-time rendering* just draws to the screen as fast as possible

Rendering (cont'd.)

- Have you ever wondered why there is an FPS count in games? Like 30 or 60
- It's because the end user can't see blindingly-fast updates so the frame rate is limited to allow the processor to perform other tasks

Rendering (cont'd.)

- *Double buffering* is a technique of rendering that uses two virtual windows or screens, called buffers
  - Front and back buffers
- The front buffer is what the user sees
- The back buffer is what's going to be drawn next frame – will become the front buffer

Adding the Sprite

```
// Game.hpp
class Game
{
   public:
     Game();
     ...
   private:
     sf::Texture mTexture;
     sf::Sprite mPlayer;
     ...
};
```

Adding the Sprite (cont'd.)

```
// Game.cpp
Game::Game()
: ...
, mTexture()
, mPlayer()
{
   if (!mTexture.loadFromFile("Media/Textures/Eagle.png"))
    {
      // Handle loading error
    }
    mPlayer.setTexture(mTexture);
    mPlayer.setPosition(100.f, 100.f);
}
```

Resources

- Resources can be defined as an external component that the game loads during runtime
  - Also known as an asset
  - Most often multimedia components such as images, music or fonts
  - Generally large in size - occupy a lot of memory

Resources (cont'd.)

- Can also be scripts that describe world content, i.e. Metadata
- Also configuration files
- Whatever the case, resources are loaded from files on the hard drive
- The RAM or the Network

Resources (cont'd.)

- To load a resource in SFML, typically we use:

```
bool loadFromFile(const std::string& filename);
```

  - mTexture.loadFromFile("Media/Textures/Eagle.png");
  - mFont.loadFromFile("Media/Sansation.ttf");

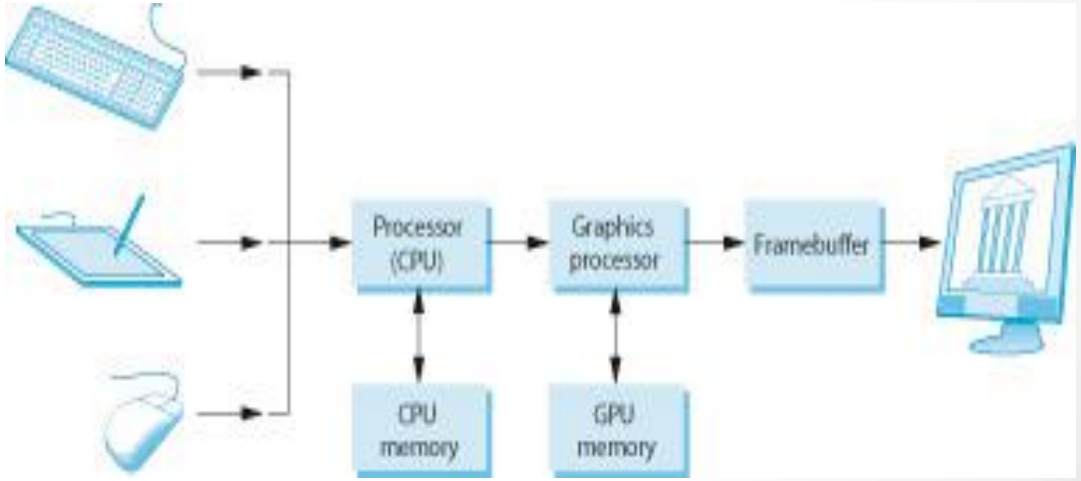
- Like most languages, the boolean return type holds whether or not the load was successful
- Checking for a successful resource load is critical in any program

Resources (cont'd.)

- loadFromStream() loads a resource using a custom sf::InputStream instance.
  - Allows the user to exactly specify the loading process
  - Use cases of user-defined streams are encrypted and/or compressed file archives.
- loadFromMemory() loads a resource from RAM
  - Useful to load resources that are directly embedded into the executable

A Graphics System

- The image we see on the output device is an array (**the raster**) of picture elements, or pixels produced by graphics system
- All modern graphics systems are raster based
- Every pixel corresponds to a location in the image
- Collectively, the pixels are stored in a part of memory called the **framebuffer**.
- frame buffer depth or precision
  - 1-bit-deep => Two colors
  - 8-bit deep => 256 colors
  - Full color => 24 bit or more



Sprite

- Even modern 2D games are made using vertices.
- The 2D sprites are made up of two triangles to form a square.
- This is often referred to as a quad.
- The quad is given a texture and it becomes a sprite.
- Two-dimensional games use a special projection transformation that ignores all the 3D data in the vertices, as it's not required for a 2D game.
- Sprites are 2D bitmaps that are drawn directly to a render target without using the pipeline for transformations, lighting or effects. Sprites are commonly used to display information such as health bars, number of lives, or text such as scores. Some games, especially older games, are composed entirely of sprites.

Textures

- Textures in SFML are represented as graphical images in the `sf::Texture` class
  - Stored as an array of pixels in the graphics card
  - Each pixel is a 32 bit RGBA value
  - Can be drawn to the screen with `sf::Sprite`

- A sprite can be part of a texture, so we consider it lightweight
- A texture is the source image for a sprite

Images

- A container for pixel values using the `sf::Image` class
  - Stores its pixels in RAM instead of video memory
  - Capable of saving the stored image back to a file.
  - So we can manipulate single pixels
  - sf::Texture loads the data using an intermediate sf::Image

- Some restrictions
  - To display a sf::Image, First have to convert it to `sf::Texture` and then create a `sf::Sprite` that refers to it
  - If you don't need per pixel access, use texture

Fonts

- To load a character font for use and manipulation, use the `sf:Font` class
- Supports many formats including the common true type fonts (TTF) and open type fonts (OTF)

- To display text on screen, use `sf::Text`
- Like the texture-sprite relationship, the font is the source for many text instances

Open GL

- Every computer has special graphics hardware that controls what you see on the screen.
- OpenGL tells this hardware what to do.
- The Open Graphics Library is one of the oldest, most popular graphics libraries game creators have.
- It was developed in 1992 by Silicon Graphics Inc. (SGI) and used for GLQuake in 1997.
- The GameCube,Wii, PlayStation, and the iPhone all use OpenGL.
- Cross-platform standard: OpenGL today is supported on all platforms

Vertex

- The basic unit in OpenGL is the vertex. A vertex is a point in space.
- Extra information can be attached to these points—how it maps to a texture, if it has a certain weight or color—but the most important piece of information is its position.

- Games spend a lot of their time sending OpenGL vertices or telling OpenGL to move vertices in certain ways.
- The game may first tell OpenGL that all the vertices it's going to send are to be made into triangles. In this case, for every three vertices OpenGL receives, it will attach them together with lines to create a polygon, and it may then fill in the surface with a texture or color.

Pipeline

- Modern graphics hardware is very good at processing vast sums of vertices, making polygons from them and rendering them to the screen.
- This process of going from vertex to screen is called the pipeline.
- The pipeline is responsible for positioning and lighting the vertices, as well as the projection transformation.

Pipeline

- The pipeline has become programmable.
- Programs can be uploaded to the graphics card.
- There are few steps in the pipeline.
- All the steps could be applied in parallel.
- Each vertex can pass through a particular stage at the same time, provided the hardware supports it.
- This is what makes graphics cards so much faster than CPU processing.

Shaders

- A program that operates on the graphics card and applies effects to rendered assets
- SFML builds upon OpenGL, so it uses GLSL (OpenGL Shading Language)
  - SFML supports Vertex shaders and fragment/pixel shaders
    - Vertex Shaders: affects geometry of objects in the scence
    - Fragment shaders: manipulate pixel of the scene
- An SFML shader or `sf::Shader` can be created from a string that contains the GLSL code of the source shader

Sounds

- SFML contains a `sf::SoundBuffer` class used to store sound effects
  - 16 bit audio samples
- `sf::Sound` is the class that actually plays audio from the sound buffer
  - Can be played, paused and stopped and have their volume and pitch modified

Sounds

- SFML contains a `sf::SoundBuffer` class used to store sound effects
  - 16 bit audio samples
- `sf::Sound` is the class that actually plays audio from the sound buffer
  - Can be played, paused and stopped and have their volume and pitch modified

Sound (cont'd.)

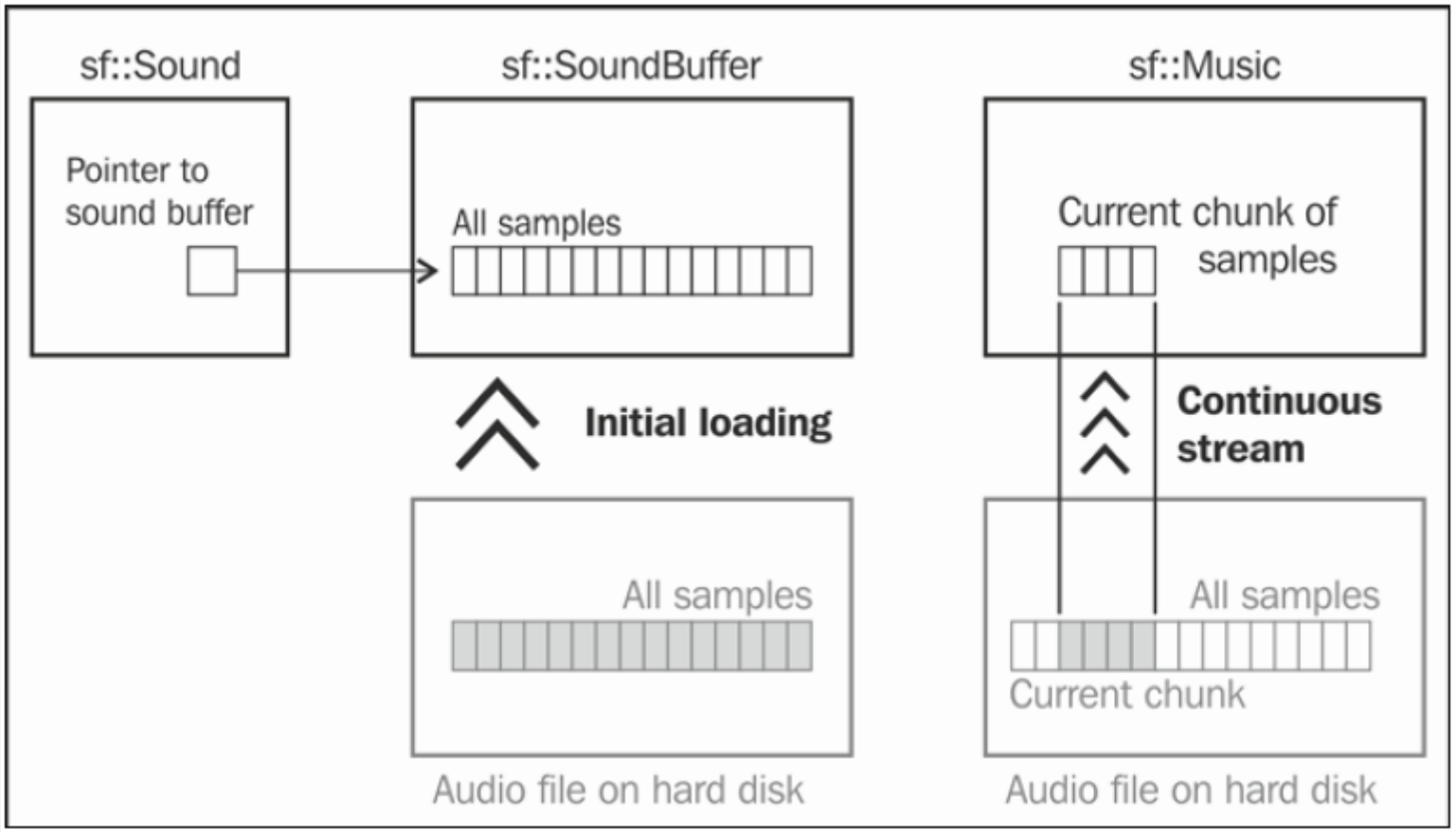
- Valid formats are WAV, OGG, AIFF and more
  - See http://www.sfml-dev.org/faq.php#audio-formats for the full list

- **SFML does NOT support MP3s because of their restrictive license**

Sound (cont'd.)

- Music in SFML is played by the sf::Music class
- Does not use the sound buffer class
- Streams the music, i.e., it loads small chunks continuously
- The difference between the sound buffer and music is shown on the next slide

Sound (cont'd.)


sf::Sound
sf::SoundBuffer
sf::Music
Pointer to sound buffer
All samples
Current chunk of samples
Initial loading
Continuous stream
All samples
All samples
Current chunk
Audio file on hard disk
Audio file on hard disk

Using Resources

- Game entities like the player and enemies are represented by sprites and text
  - They access the source files but do not own them
- All resources must remain in scope for however long they are needed in the game
- Game entities are separated from any sounds that are associated with them
  - For example, we want to play the enemy death sound even if the enemy object has been removed

Using Resources (cont'd.)

- Typically, you want to load the resource before you use it – for example, upon the start of a game or new level
  - So the game performance is not affected
- Similarly, release resources at the end of a level or game
- You will have a class that manages that functionality
  - RAII or Resource Acquisition Is Initialization
  - http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

Using Resources (cont'd.)

- Choosing the right data structure to hold the resources is important as well
  - We're going to use a map (`std::map`) for our example as we're not going to change its size
  - We want to avoid reallocating the structure
  - We're also going to be using an enumeration to act as our key type for our textures

- Let's start creating our containers

Working with Textures

```
class TextureHolder
{
   public:
   void load(Textures::ID id, const std::string& filename);

   private:
   std::map<Textures::ID,std::unique_ptr<sf::Texture>> mTextureMap;
};

void TextureHolder::load(Textures::ID id, const std::string& filename)
{
   std::unique_ptr<sf::Texture> texture(new sf::Texture());
   texture->loadFromFile(filename);

   mTextureMap.insert(std::make_pair(id, std::move(texture)));
}
```

Type Inference

- Woah, hold on there... what was that `auto` type?
  - New C++ 11 feature
  - The `auto` keyword deduces what the variable should be based on the left side of the assignment
  - Articles below explain it more:
    - http://www.cprogramming.com/c++11/c++11-auto-decltype-return-value-after-function.html
    - http://en.wikipedia.org/wiki/C%2B%2B11

Accessing the Textures

- Now we want to grant access to our textures via a few accessors/getters:

```
sf::Texture& get(Textures::ID id);

const sf::Texture& get(Textures::ID id) const;

sf::Texture& TextureHolder::get(Textures::ID id)
{
   auto found = mTextureMap.find(id);
   return *found->second;
}
```

A const method can be called on a const object

```
class CL2 {
 public: void const_method() const;
void method();
private: int x;
 };

const CL2 co;
CL2 o;
co.const_method(); // legal
co.method(); // illegal, can't call regular method on const object
o.const_method(); // legal, can call const method on a regulard object
o.method(); // legal
```

New TextureHolder

- Our TextureHolder class now looks like this:

```
class TextureHolder
{
     public:
     void load(Textures::ID id, const std::string& filename);
     sf::Texture& get(Textures::ID id);
     const sf::Texture& get(Textures::ID id) const;

     private:
     std::map<Textures::ID,std::unique_ptr<sf::Texture>> mTextureMap;
};
```

- And can be used in the following manner:

```
TextureHolder textures;
textures.load(Textures::Airplane, "Media/Textures/Airplane.png");

sf::Sprite playerPlane;
playerPlane.setTexture(textures.get(Textures::Airplane));
```

Error Handling

- The program could encounter many errors in the program, and it is important to account for them
- So let's add some of the more common approaches

```
if (!texture->loadFromFile(filename))
   throw std::runtime_error("TextureHolder::load -
   Failed to load "+ filename);
```

Error Handling (cont'd.)

- Let's have a look at the full `load` method:

```
void TextureHolder::load(Textures::ID id, const std::string&
   filename)
{
   std::unique_ptr<sf::Texture> texture(new sf::Texture());

   if (!texture->loadFromFile(filename))
        throw std::runtime_error("TextureHolder::load - Failed to
        load " + filename);

   auto inserted = TextureMap.insert(std::make_pair(id,
   std::move(texture)));

   assert(inserted.second);
}
```

Error Handling (cont'd.)

- We can add an `assert` to the `get` method too:

```
sf::Texture& TextureHolder::get(Textures::ID id)
{
   auto found = mTextureMap.find(id);
   assert(found != mTextureMap.end());
   return *found->second;
}
```

Error Handling (cont'd.)

- We can add an `assert` to the `get` method too:

```
sf::Texture& TextureHolder::get(Textures::ID id)
{
   auto found = mTextureMap.find(id);
   assert(found != mTextureMap.end());
   return *found->second;
}
```

Generalized Template

- Our `TextureHolder` is great, but can we create others for different resources?
  - Yes, but we can do something even better!
  - Alter `TextureHolder` and make it a general class
  - We can call it `ResourceHolder`
  - It will be a template class with two template parameters
    - The type of resource and an ID type for resource access

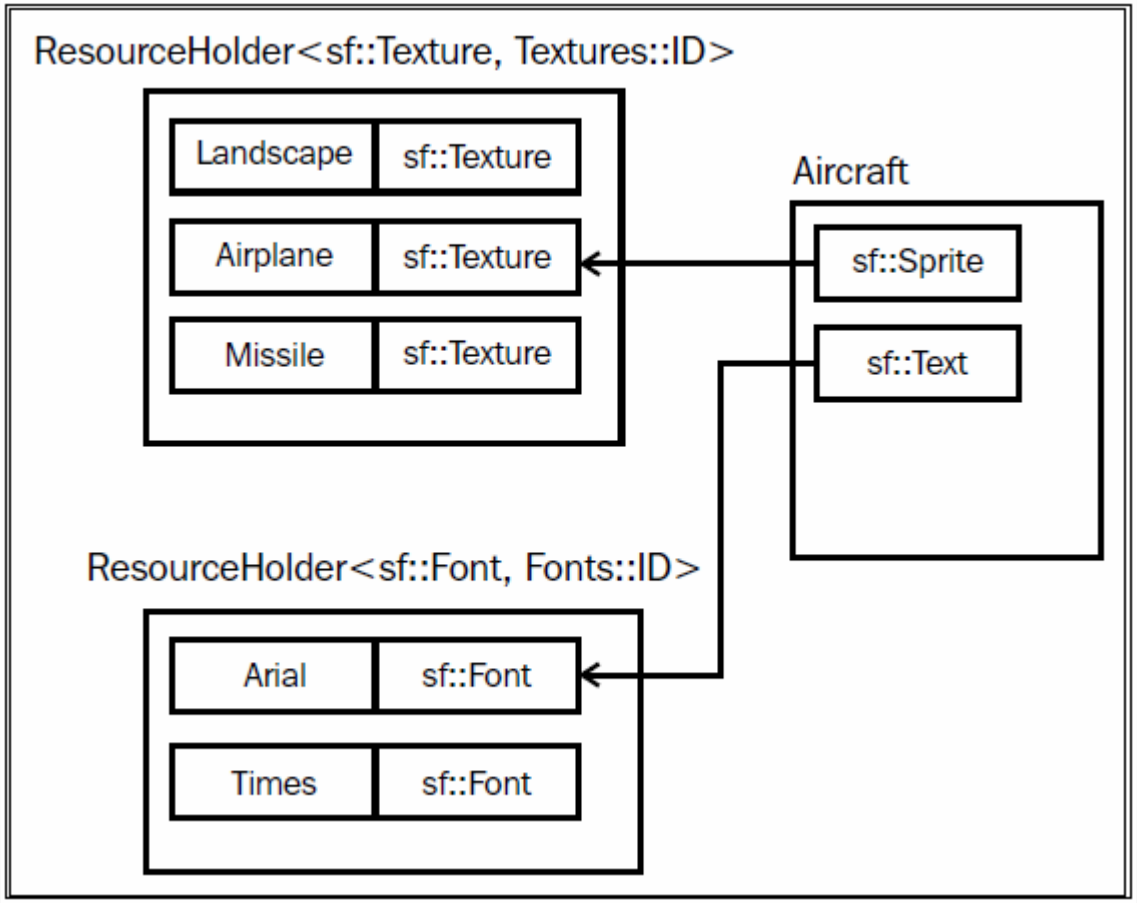
ResourceHolder Class

```
template <typename Resource, typename Identifier>
class ResourceHolder
{
    public:
      void load(Identifier id, const std::string& filename);
      Resource& get(Identifier id);
      const Resource& get(Identifier id) const;
    private:
      std::map<Identifier,
      std::unique_ptr<Resource>> mResourceMap;
};
```

`load` Method

```
template <typename Resource, typename Identifier>
void ResourceHolder<Resource, Identifier>::load(Identifier id,
const std::string& filename)
{
   std::unique_ptr<Resource> resource(new Resource());
   if (!resource->loadFromFile(filename))
        throw std::runtime_error("ResourceHolder::load - Failed to
        load " + filename);
   auto inserted = mResourceMap.insert(std::make_pair(id,
   std::move(resource)));
   assert(inserted.second);
}
```

Visualizing ResourceHolder



Appendix A

- Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

```
// function template
#include <iostream>
using namespace std;

template <class T>
T GetMax (T a, T b) {
  T result;
  result = (a>b)? a : b;
  return (result);
}
int main () {
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax<int>(i,j);
  n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```