Week 13

Cube Mapping, Normal Mapping, and Shadow Maps

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.

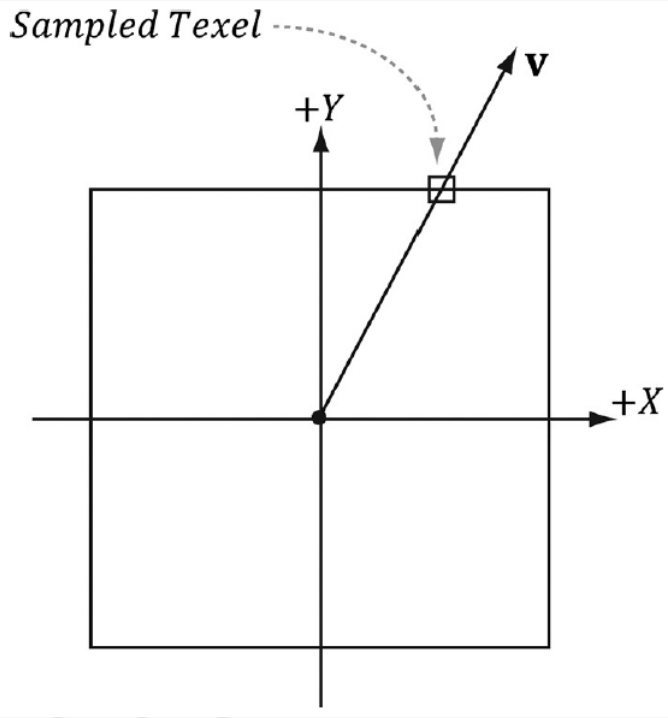
4. To understand how we can texture a sphere with cube maps to simulate a sky and distant mountains.

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 a 3D *lookup* vector **v** 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 with some coordinate system. We shoot a vector **v** from the origin. The texel **v** intersects is the sampled texel. In this illustration, **v** intersects the cube face corresponding to the *+Y* axis.

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.

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.

```
TextureCube gCubeMap : register(t0);

// in pixel shader

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

ENVIRONMENT MAPS

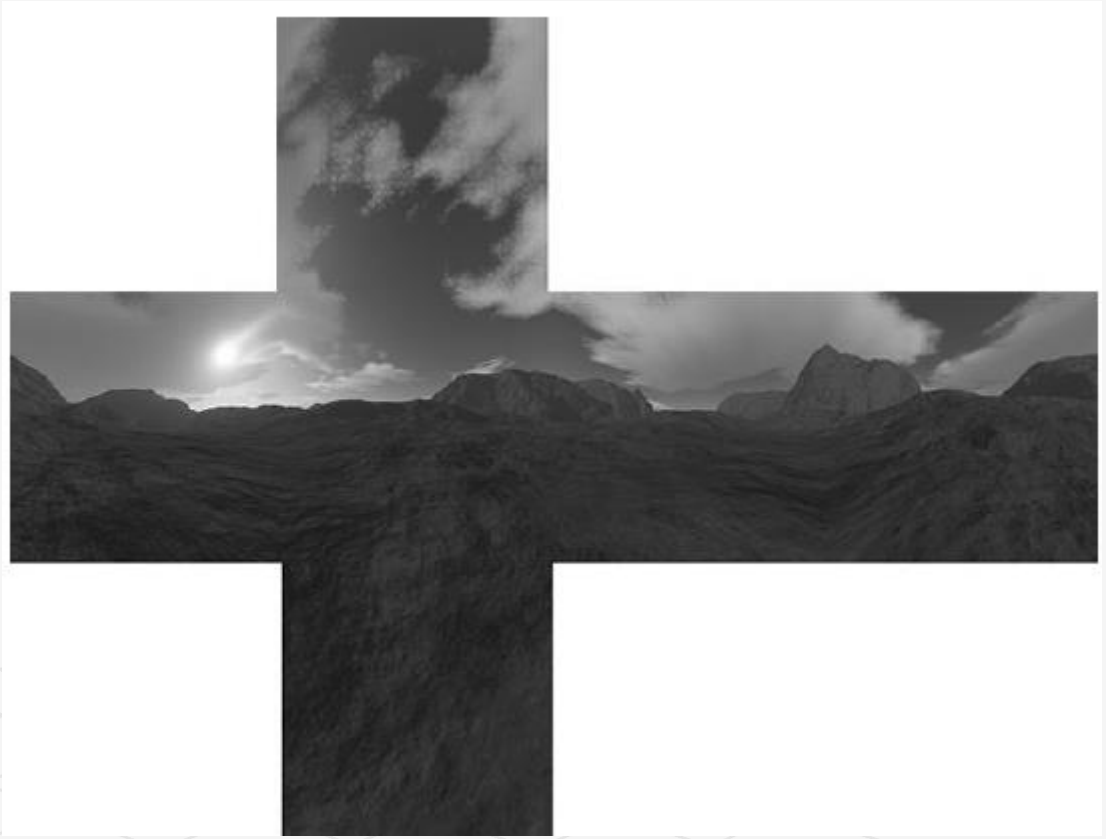
The primary application of cube maps is *environment mapping*.

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 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.

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.

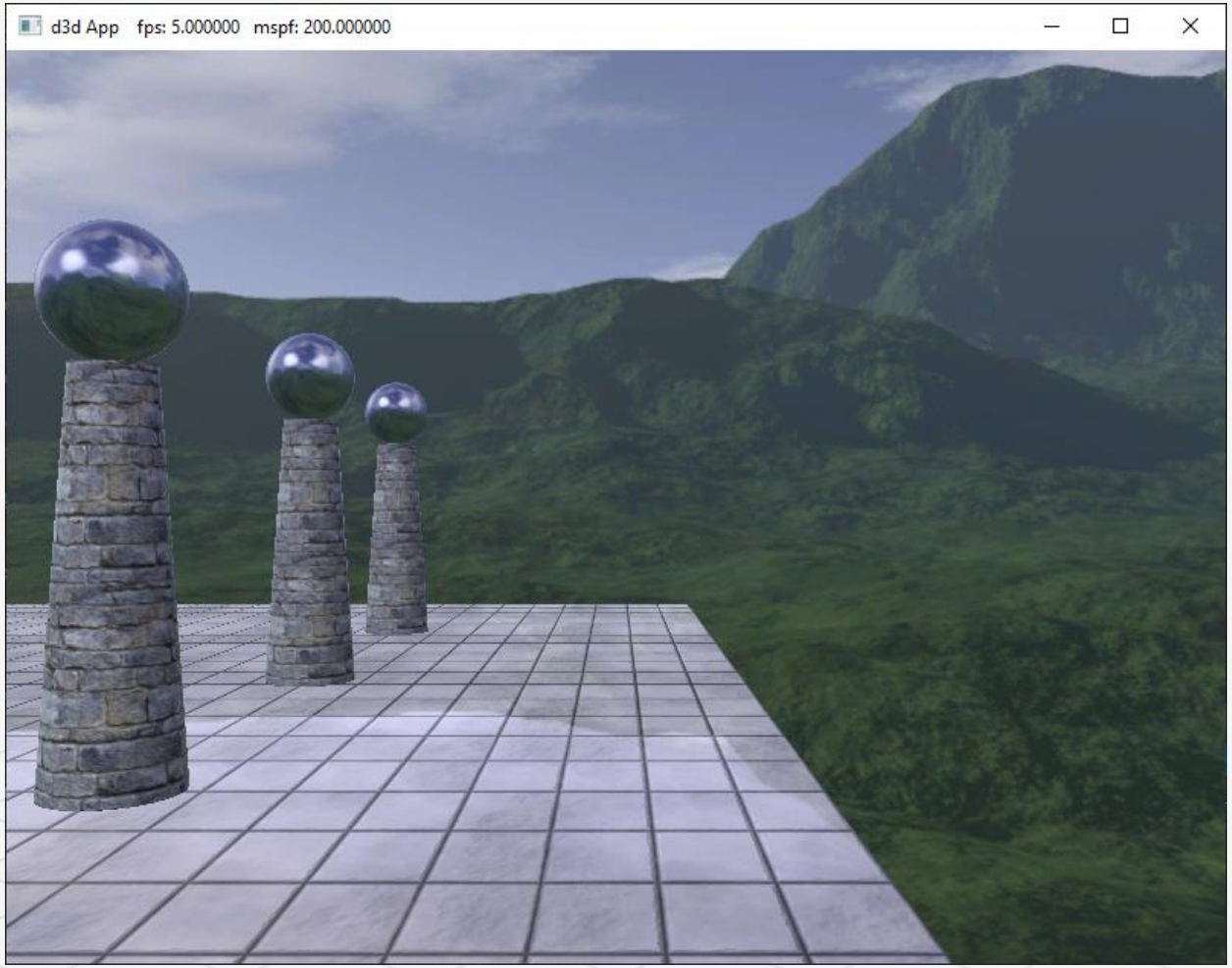
For outdoor environment maps, the program *Terragen:*

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. There is a nice* Terragen *script on the web*

https://developer.valvesoftware.com/wiki/Skybox_(2D)_with_Terragen that will use 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 (taken from the *texassemble* documentation):

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

NVIDIA provides Photoshop plugins for saving .DDS and cubemaps in Photoshop; see *http://developer.nvidia.com/nvidia-texture-tools-adobephotoshop*

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);
    }
}
```

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:

```
// 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);

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

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

TEXTURING A SKY

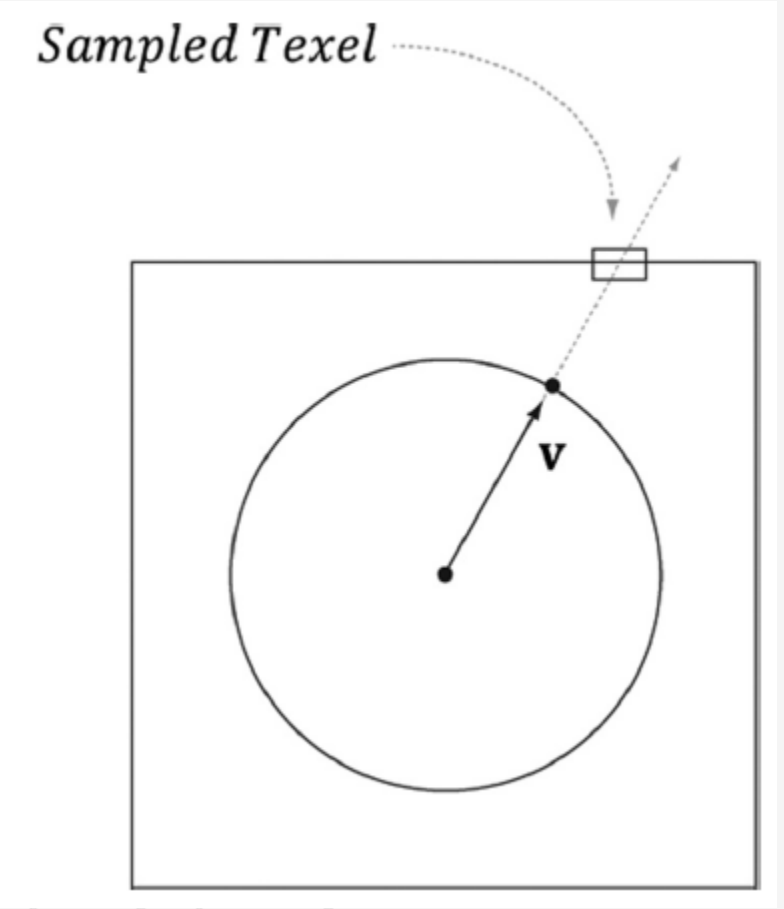
We create a large sphere that surrounds the entire scene.

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. Then 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);
}
```

SetPipelineState

Drawing the sky requires different shader programs, and hence a new PSO. Therefore, we draw the sky as a separate layer in our drawing code:

```
// 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(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"])));
```

MODELING REFLECTIONS

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

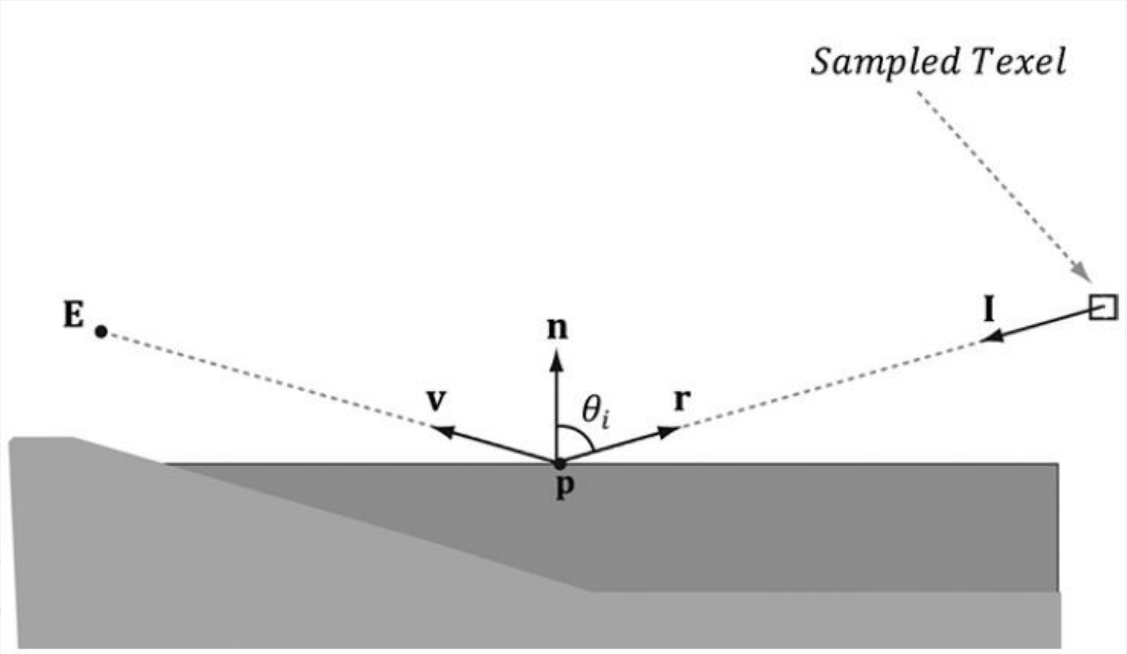
By specular reflections, we mean that we are just going to look at the light that is reflected off a surface due to the Fresnel effect.

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

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

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(gsamAnisotropicWra
p, 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.

//We compute the reflection vector per-pixel and then use it to sample the environment map:

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


//we need to apply the Fresnel effect, which 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;

}
```

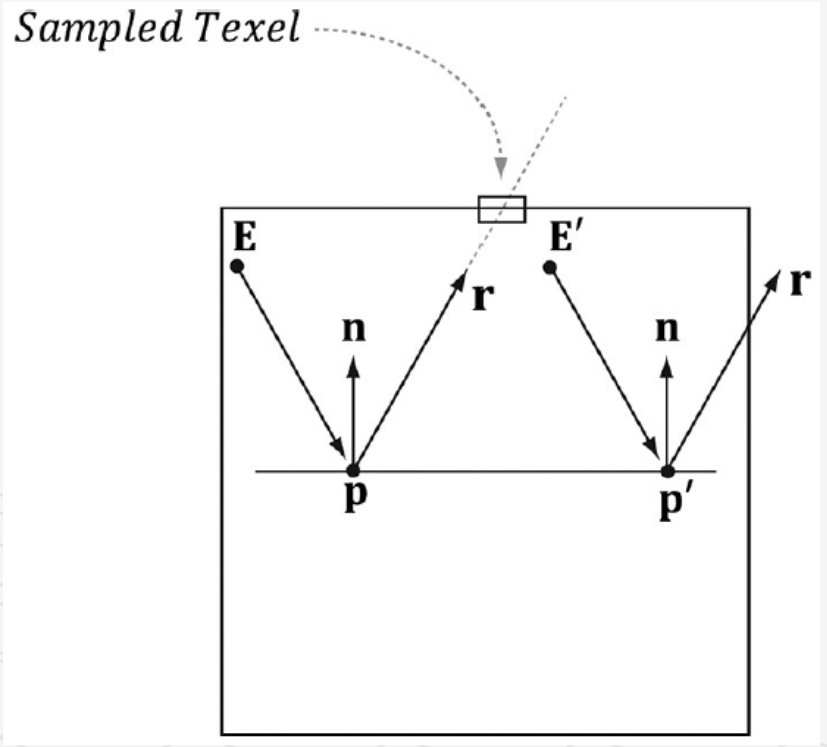
Flat Surfaces

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

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. The reflection vector does not tell the whole story, as it does not incorporate position; 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.



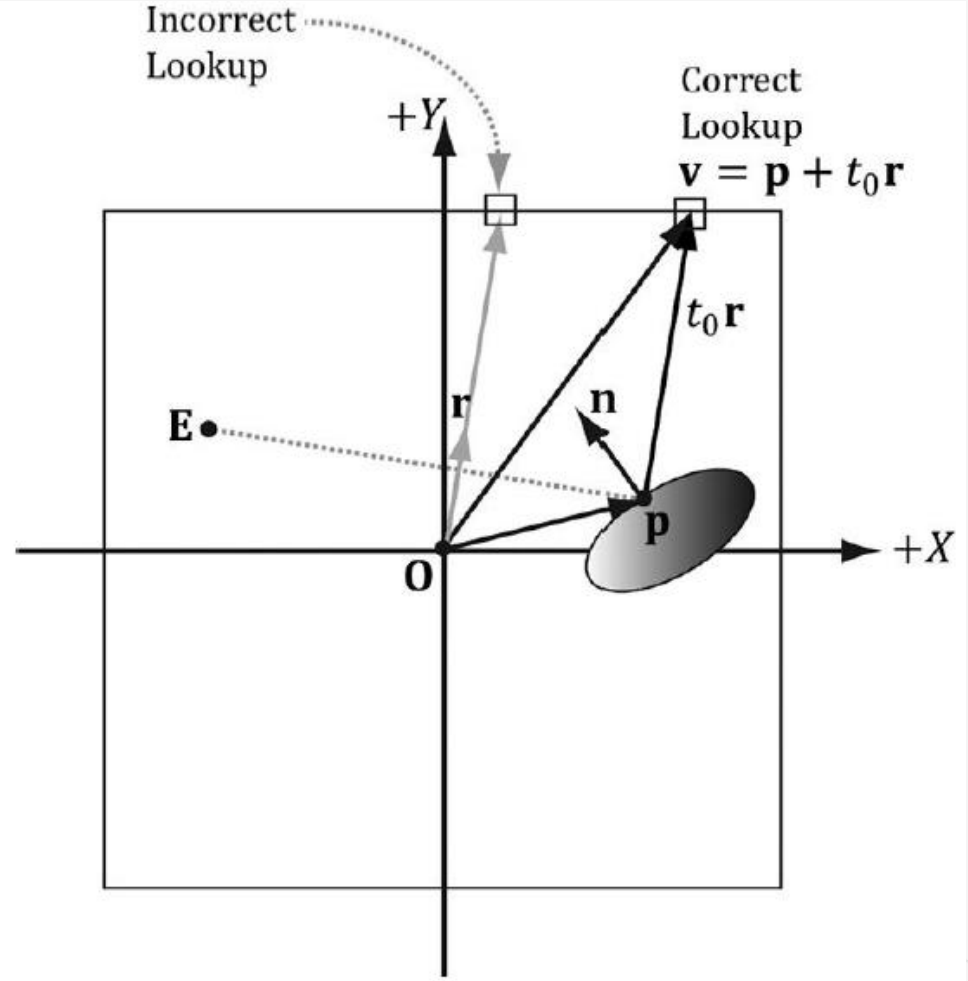
Flat Surfaces

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 steading of using the reflection vector **r** for the cube map lookup, we use the intersection point **v** = **p** + *t*0**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.



DYNAMIC CUBE MAPS

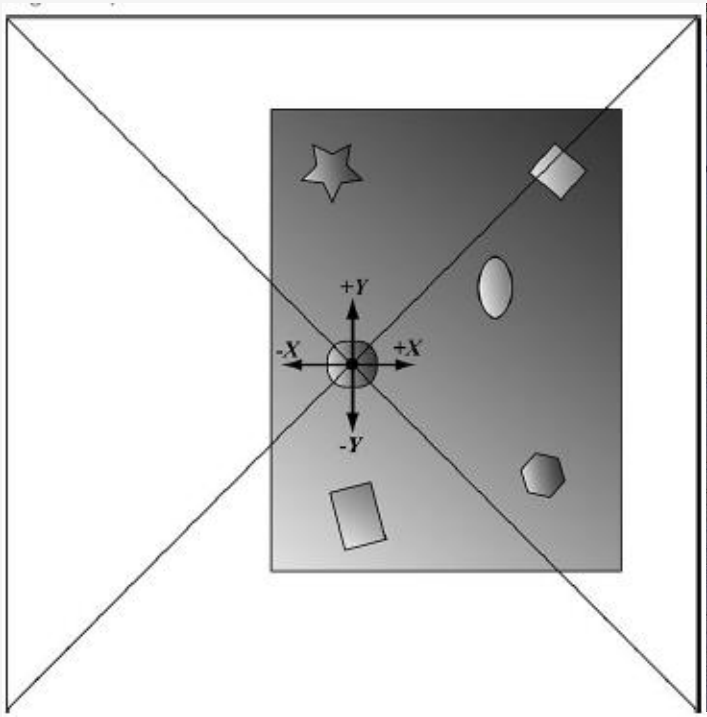
So far we have described static cube maps, where the images stored in the cube map are premade and fixed.

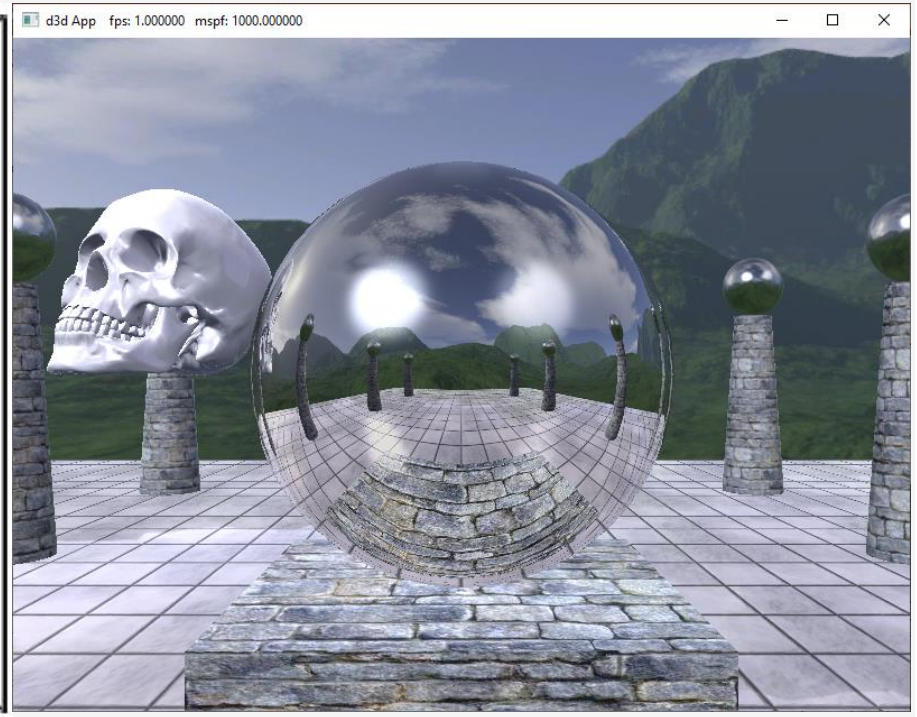
With a pre-generated cube map, you cannot capture animated objects, which means we cannot reflect animated objects.

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. Below is the method that 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 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, and we also need to create a render target view to eachelement 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. This is easy enough, as we just 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:

```
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));
}
```

DynamicCubeMapApp::Update

Finally, after we rendered the scene to the cube map, we set our main render targets and draw the scene as normal, but with the dynamic cube map applied to the center sphere:

```
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 section, we redrew the scene six times to generate the cube map—once for each cube map face.

There is a Direct3D 10 sample called "CubeMapGS," which uses 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

D3D10_RENDER_TARGET_VIEW_DESC DescRT;

DescRT.Format = dstex.Format;

DescRT.ViewDimension =

D3D10_RTV_DIMENSION_TEXTURE2DARRAY;

DescRT.Texture2DArray.FirstArraySlice = 0;

DescRT.Texture2DArray.ArraySize = 6;

DescRT.Texture2DArray.MipSlice = 0;

V_RETURN(pd3dDevice->CreateRenderTargetView(

g_pEnvMap, &DescRT, &g_pEnvMapRTV));
```

CubeMapGS

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

```
// Create the depth stencil view for the entire cube

D3D10_DEPTH_STENCIL_VIEW_DESC DescDS;

DescDS.Format = DXGI_FORMAT_D32_FLOAT;

DescDS.ViewDimension = D3D10_DSV_DIMENSION_TEXTURE2DARRAY;

DescDS.Texture2DArray.FirstArraySlice = 0;

DescDS.Texture2DArray.ArraySize = 6;

DescDS.Texture2DArray.MipSlice = 0;

V_RETURN(pd3dDevice->CreateDepthStencilView(

g_pEnvMapDepth, &DescDS, &g_pEnvMapDSV));
```

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

```
ID3D10RenderTargetView* aRTViews[1] = { g_pEnvMapRTV };

pd3dDevice->OMSetRenderTargets(sizeof(aRTViews) / sizeof(aRTViews[0]), aRTViews,
g_pEnvMapDSV);
```

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.

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 prebaked 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.

Normal Mapping

Objectives:

1. To understand why we need normal mapping.

2. To discover how normal maps are stored.

3. To learn how normal maps can be created.

4. To find out the coordinate system the normal vectors in normal maps are stored relative to and how it relates to the object space coordinate system of a 3D triangle.

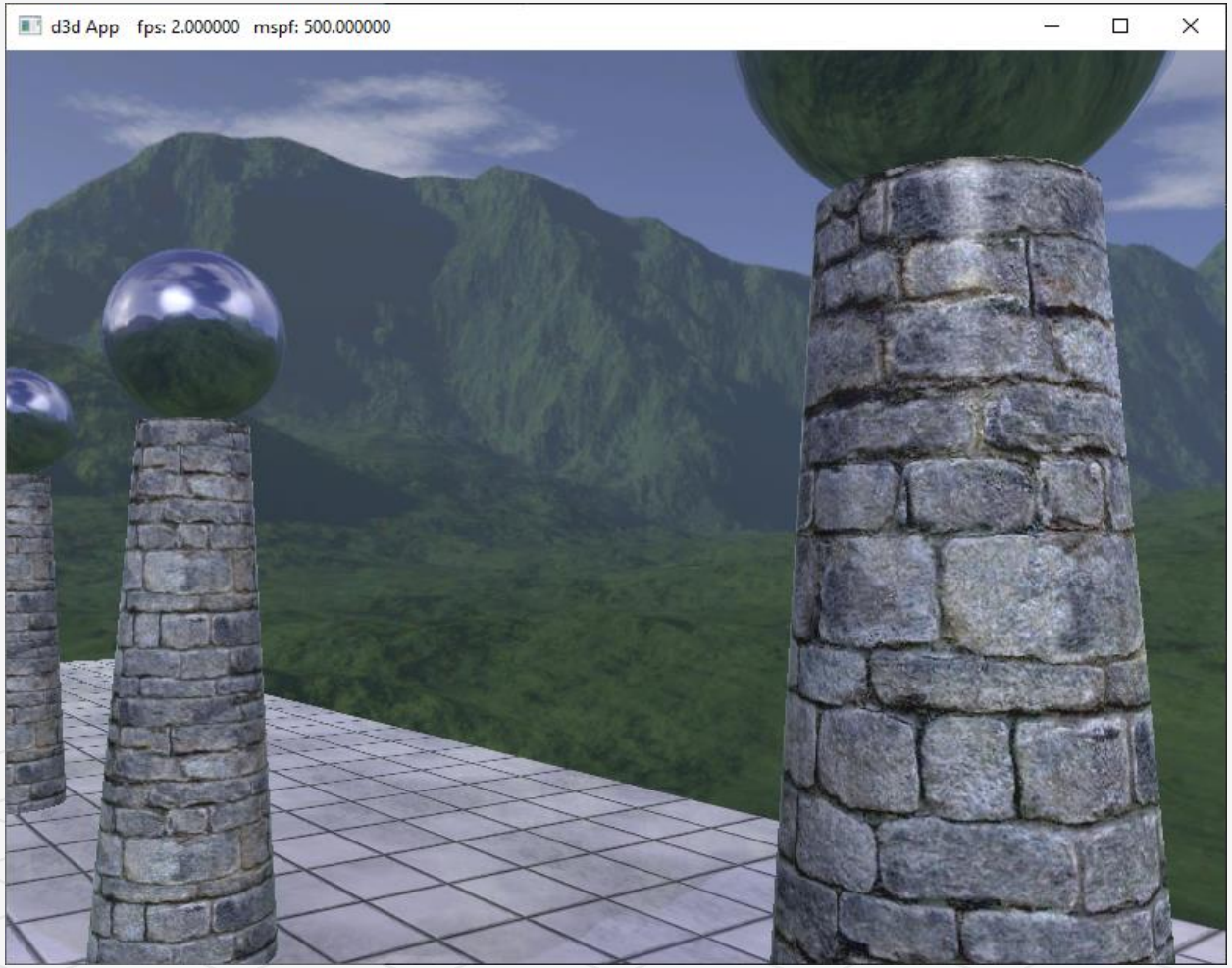
5. To learn how to implement normal mapping in a vertex and pixel shader.

MOTIVATION

The specular highlights on the cone shaped columns do not look right—they look unnaturally smooth compared to the bumpiness of the brick texture.

Because the underlying mesh geometry is smooth, and we have merely applied the image of bumpy bricks over the smooth cylindrical surface.

The lighting calculations are performed based on the mesh geometry (in particular, the interpolated vertex normals), and not the texture image.



NORMAL MAPS

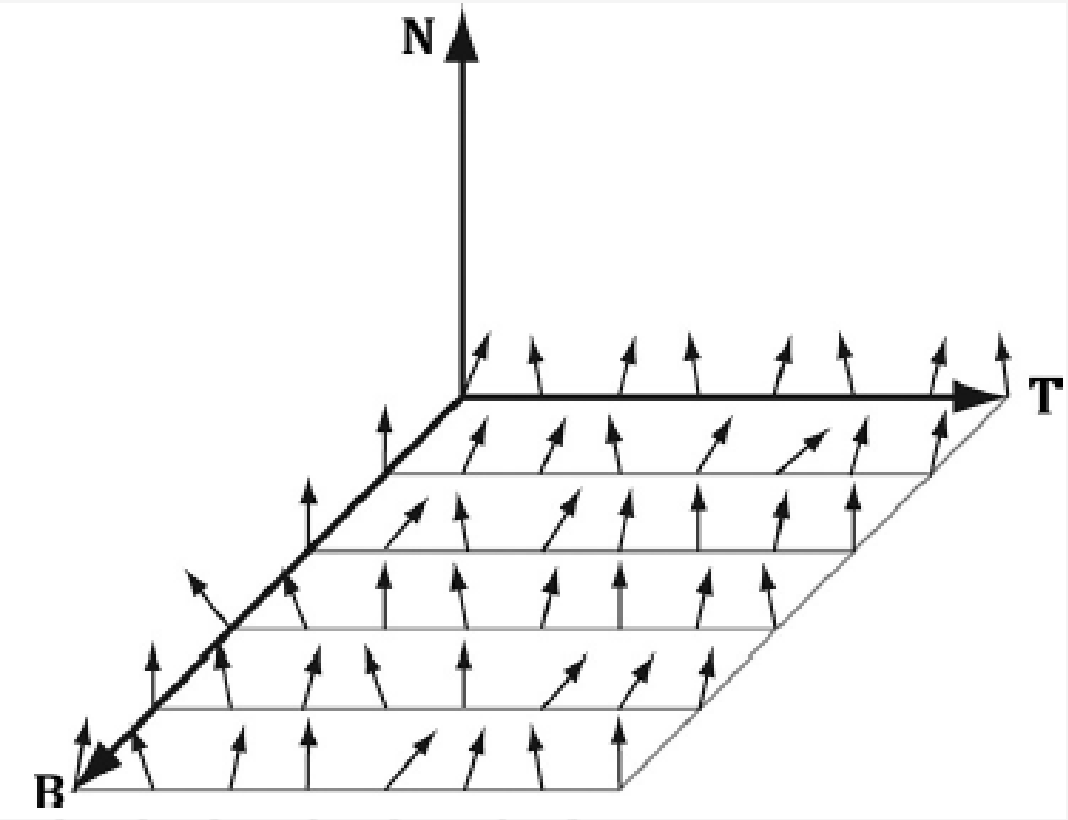
A *normal map* is a texture, but instead of storing RGB data at each texel, we store a compressed *x*-, *y*-, and *z*-coordinates in the red, green, and blue components, respectively. These coordinates define a normal vector;

A normal map stores a normal vector at each pixel.

Normals stored in a normal map relative to a texture space coordinate system defined by the vectors **T** (*x*-axis), **B** (*y*-axis), and **N** (*z*-axis). The **T** vector runs right horizontally to the texture image; the **B** vector runs down vertically to the texture image; and **N** is orthogonal to the texture plane.

The **T**, **B**, and **N** vectors are commonly referred to as the *tangent*, *binormal* (or *bitangent*), and *normal* vectors, respectively.

For illustration, we will assume a 24-bit image format, which reserves a byte [0-255] for each color component



Compression Texture Coordinates

How do we compress a unit vector [-1,1] into this 24-bit or 32 bit format [0-255][0-255][0-255][0-255]?

If we shift and scale this range to [0, 1] and multiply by 255 and truncate the decimal, the result will be an integer in the range 0-255.

if x is a coordinate in the range $[-1, 1]$, then the integer part of $f(x)$

$$f(x) = (0.5x + 0.5).255$$

So to store a unit vector in 24-bit image, we just apply f to each coordinate and write the coordinate to the corresponding color channel in the texture map.

How to reverse the compression process; that is, given a compressed texture coordinate in the range 0-255, how can we recover its true value in the interval $[-1, 1]$? Invert the function f.

$$f - 1(x) = \frac{2x}{255} - 1$$

We will not have to do the compression process ourselves, as we will use a Photoshop plug-in to convert images to normal maps. However, when we sample a normal map in a pixel shader, we will have to do part of the inverse process to un-compress it. When we sample a normal map in a shader like this:

```
float3 normalT = gNormalMap.Sample(gTriLinearSam, pin.Tex);
```

The color vector normalT will have normalized components (r, g, b) such that $0 \le r, g, b \le 1$.

```
// Uncompress each component from [0,1] to [-1,1].

normalT = 2.0f * normalT - 1.0f;
```

The Photoshop plug-in is available at https://developer.nvidia.com/nvidia-texture-tools-adobe-photoshop

There are other tools available for generating normal maps such as http://www.crazybump.com/ and

http://shadermap.com/home/

TEXTURE/TANGENT SPACE

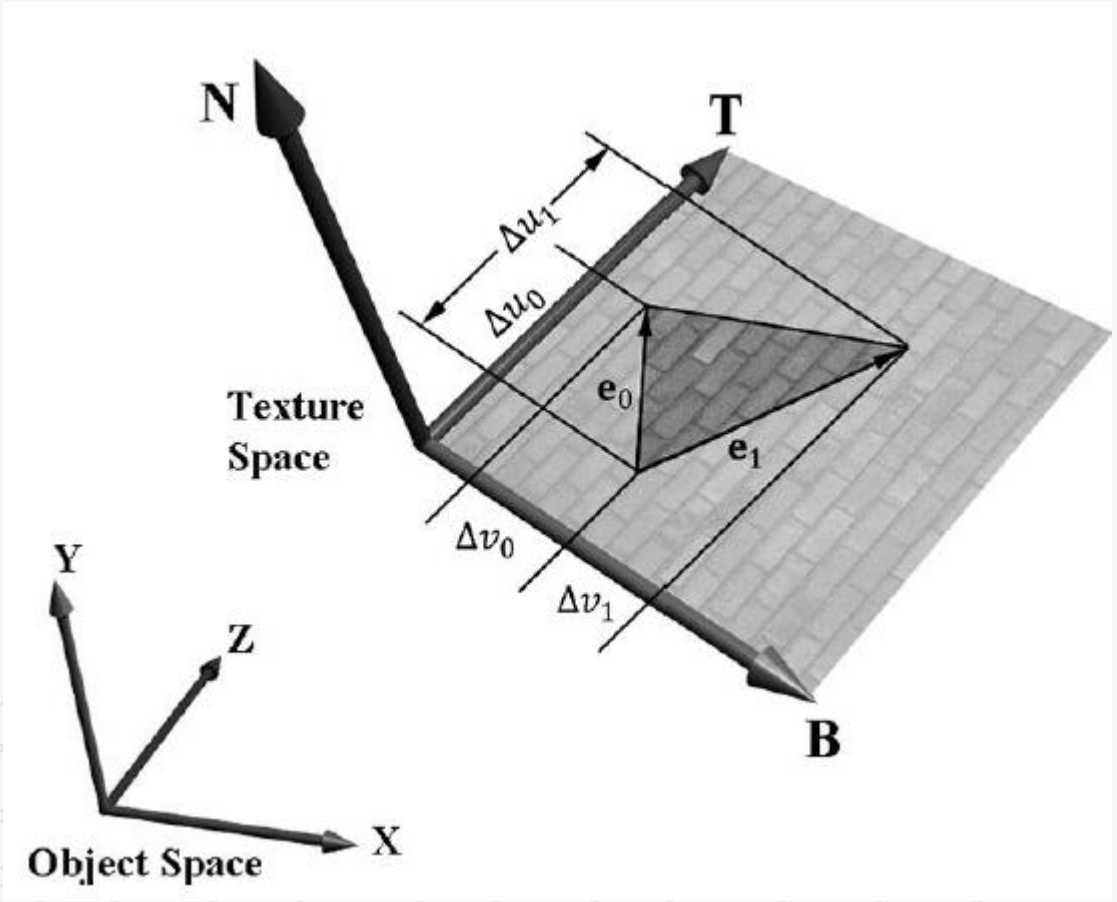
Consider a 3D texture mapped triangle.

The relationship between the texture space of a triangle and the object space.

The 3D tangent vector **T** aims in the *u*-axis direction of the texturing coordinate system.

The 3D tangent vector **B** aims in the *v*-axis direction of the texturing coordinate system.

Figure shows how the texture space axes relate to the 3D triangle: they are tangent to the triangle and lie in the plane of the triangle.



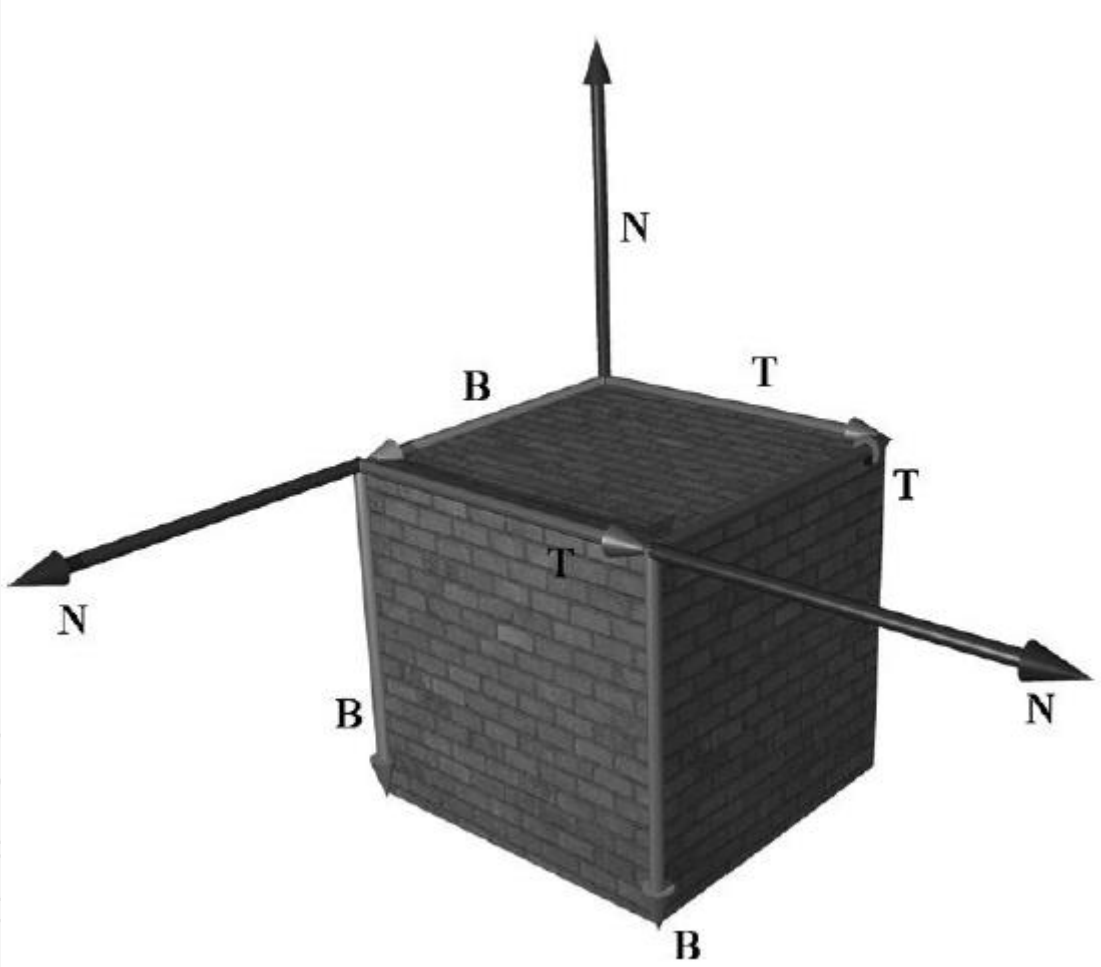
TEXTURE/TANGENT SPACE

The texture coordinates of the triangle are, of course, relative to the texture space coordinate system.

Incorporating the triangle face normal **N**, we obtain a 3D *TBN-basis* in the plane of the triangle that we call *texture space* or *tangent space*.

Note that the tangent space generally varies from triangle-to-triangle.

The normal vectors in a normal map are defined relative to the texture space. But our lights are defined in world space. In order to do lighting, the normal vectors and lights need to be in the same space. So our first step is to relate the tangent space coordinate system with the object space coordinate system the triangle vertices are relative to.



VERTEX TANGENT SPACE

If we use this texture space for normal mapping, we will get a triangulated appearance since the tangent space is constant over the face of the triangle.

We specify tangent vectors per vertex, and we do the same averaging trick that we did with vertex normals to approximate a smooth surface:

1. The tangent vector **T** for an arbitrary vertex **v** in a mesh is found by averaging the tangent vectors of every triangle in the mesh that shares the vertex **v**.

2. The bitangent vector **B** for an arbitrary vertex **v** in a mesh is found by averaging the bitangent vectors of every triangle in the mesh that shares the vertex **v**.

After averaging, the TBN-bases will generally need to be orthonormalized, so that the vectors are mutually orthogonal and of unit length.

This is usually done using the Gram-Schmidt procedure.

In our system, we will not store the bitangent vector **B** directly in memory. Instead, we will compute **B** = **N** × **T** when we need **B**, where **N** is the usual averaged vertex normal. Hence, our vertex structure looks like this:

```
struct VertexIn

{

float3 PosL    : POSITION;

float3 NormalL : NORMAL;

float2 TexC    : TEXCOORD;

float3 TangentU : TANGENT;

};
```

GeometryGenerator compute the tangent vector **T** corresponding to the *u*-axis of the texture space. The object space coordinates of the tangent vector **T** is easily specified at each vertex for box and grid meshes

TRANSFORMING BETWEEN TANGENT SPACE AND OBJECT SPACE

we have the coordinate of the TBN-basis relative to the object space coordinate system, we can transform coordinates from tangent space to object space with the matrix:

$$\mathbf{M}_{object} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

Since this matrix is orthogonal, its inverse is its transpose. Thus, the change of coordinate matrix from object space to tangent space is:

$$\mathbf{M}_{tangent} = \mathbf{M}_{object}^{-1} = \mathbf{M}_{object}^{T} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

In our shader program, we will actually want to transform the normal vector from tangent space to world space for lighting. One way would be to transform the normal from tangent space to object space first, and then use the world matrix to transform from object space to world space:

$$n_{world} = (n_{tangent} M_{object}) M_{world}$$

So to go from tangent space directly to world space, we just have to describe the tangent basis in world coordinates, which can be done by transforming the TBN-basis from object space coordinates to world space coordinates.

$$n_{world} = n_{tangent} (M_{object} M_{world})$$

$$\mathbf{M}_{object}\mathbf{M}_{world} = \begin{bmatrix} \leftarrow \mathbf{T} \rightarrow \\ \leftarrow \mathbf{B} \rightarrow \\ \leftarrow \mathbf{N} \rightarrow \end{bmatrix} \mathbf{M}_{world} = \begin{bmatrix} \leftarrow \mathbf{T}' \rightarrow \\ \leftarrow \mathbf{B}' \rightarrow \\ \leftarrow \mathbf{N}' \rightarrow \end{bmatrix} = \begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N'_x & N'_y & N'_z \end{bmatrix}$$

where $\mathbf{T}' = \mathbf{T} \cdot \mathbf{M}_{world}$, $\mathbf{B}' = \mathbf{B} \cdot \mathbf{M}_{world}$, and $\mathbf{N}' = \mathbf{N} \cdot \mathbf{M}_{world}$

NORMAL MAPPING SHADER CODE

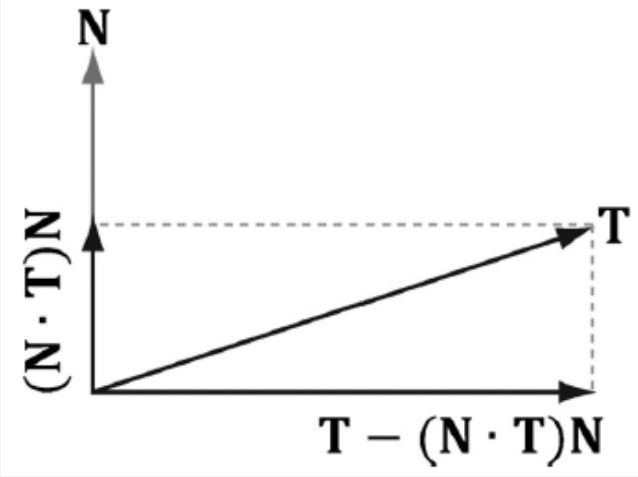
1. Create the desired normal maps from some art program or utility program and store them in an image file. Create 2D textures from these files when the program is initialized.

2. For each triangle, compute the tangent vector **T**. Obtain a per-vertex tangent vector for each vertex **v** in a mesh by averaging the tangent vectors of every triangle in the mesh that shares the vertex **v**. (In our demo, we use simply geometry and are able to specify the tangent vectors directly, but this averaging process would need to be done if using arbitrary triangle meshes made in a 3D modeling program.)

3. In the vertex shader, transform the vertex normal and tangent vector to world space and output the results to the pixel shader.

4. Using the interpolated tangent vector and normal vector, we build the TBN-basis at each pixel point on the surface of the triangle. We use this basis to transform the sampled normal vector from the normal map from tangent space to the world space. We then have a world space normal vector from the normal map to use for our usual lighting calculations.

```
//--------------------------------------------------------------------------------
-------
// Transforms a normal map sample to world space.
//--------------------------------------------------------------------------------
-------
float3 NormalSampleToWorldSpace(float3 normalMapSample, float3 unitNormalW, float3
tangentW)
{
// Uncompress each component from [0,1] to [-1,1].
float3 normalT = 2.0f*normalMapSample - 1.0f;

// Build orthonormal basis➔look at the figure
float3 N = unitNormalW;
float3 T = normalize(tangentW - dot(tangentW, N)*N);
float3 B = cross(N, T);

float3x3 TBN = float3x3(T, B, N);

// Transform from tangent space to world space.
float3 bumpedNormalW = mul(normalT, TBN);

return bumpedNormalW;
}
```



This function is used like this in the pixel shader:

```
float3 normalMapSample = gNormalMap.Sample(samLinear,pin.Tex).rgb;


float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample,pin.NormalW,
pin.TangentW);
```

bumpedNormalW vector

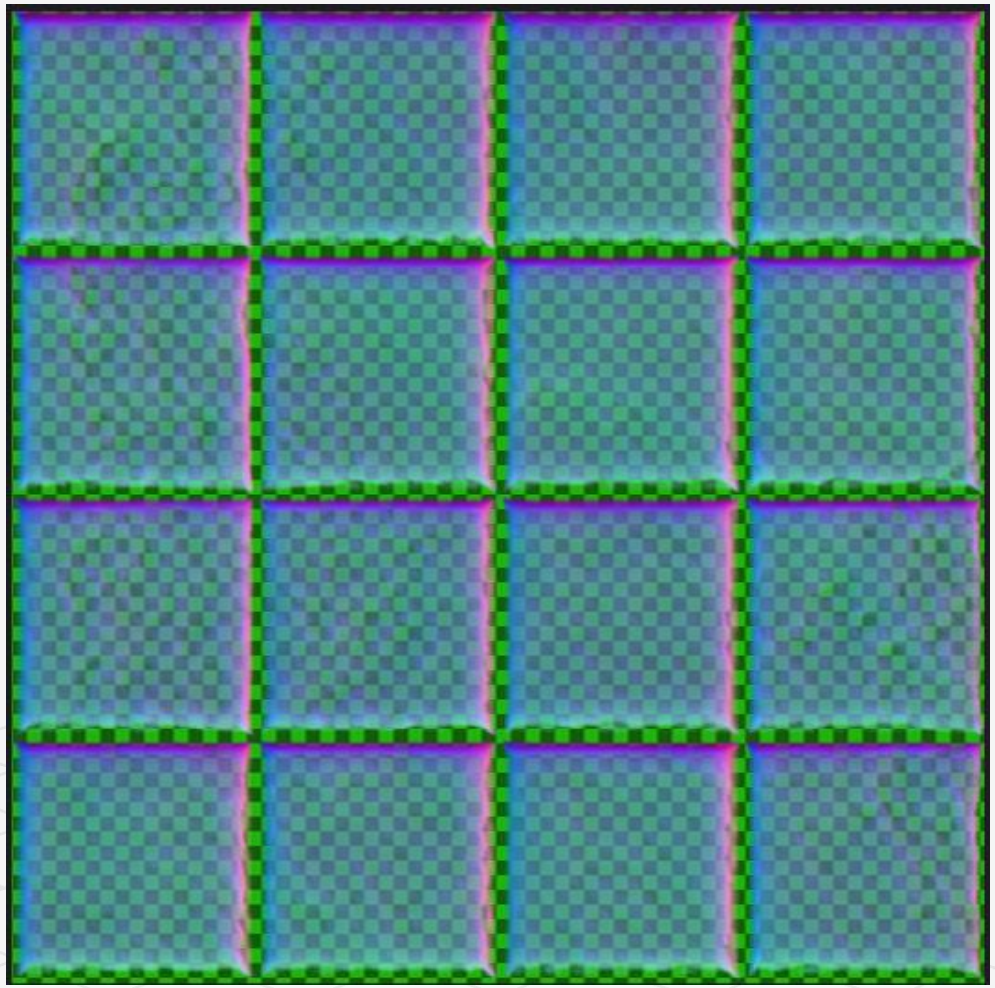
Observe that the “bumped normal” vector is use in the light calculation, but also in the reflection calculation for modeling reflections from the environment map.

```
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 diffuseMapIndex = matData.DiffuseMapIndex;
uint normalMapIndex = matData.NormalMapIndex;

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

float4 normalMapSample =
gTextureMaps[normalMapIndex].Sample(gsamAnisotropicWrap, pin.TexC);
float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample.rgb,
pin.NormalW, pin.TangentW);

// Uncomment to turn off normal mapping.
//bumpedNormalW = pin.NormalW;

// Dynamically look up the texture in the array.
diffuseAlbedo *=
gTextureMaps[diffuseMapIndex].Sample(gsamAnisotropicWrap, pin.TexC);

    // 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) * normalMapSample.a;
    Material mat = { diffuseAlbedo, fresnelR0, shininess };
    float3 shadowFactor = 1.0f;
    float4 directLight = ComputeLighting(gLights, mat, pin.PosW,
        bumpedNormalW, toEyeW, shadowFactor);

    float4 litColor = ambient + directLight;

// Add in specular reflections.
float3 r = reflect(-toEyeW, bumpedNormalW);
float4 reflectionColor = gCubeMap.Sample(gsamLinearWrap, r);
float3 fresnelFactor = SchlickFresnel(fresnelR0, bumpedNormalW, r);
    litColor.rgb += shininess * fresnelFactor * reflectionColor.rgb;

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

    return litColor;
}
```

Shininess

In addition, in the alpha channel of the normal map we store a shininess mask, which controls the shininess at a per-pixel level.

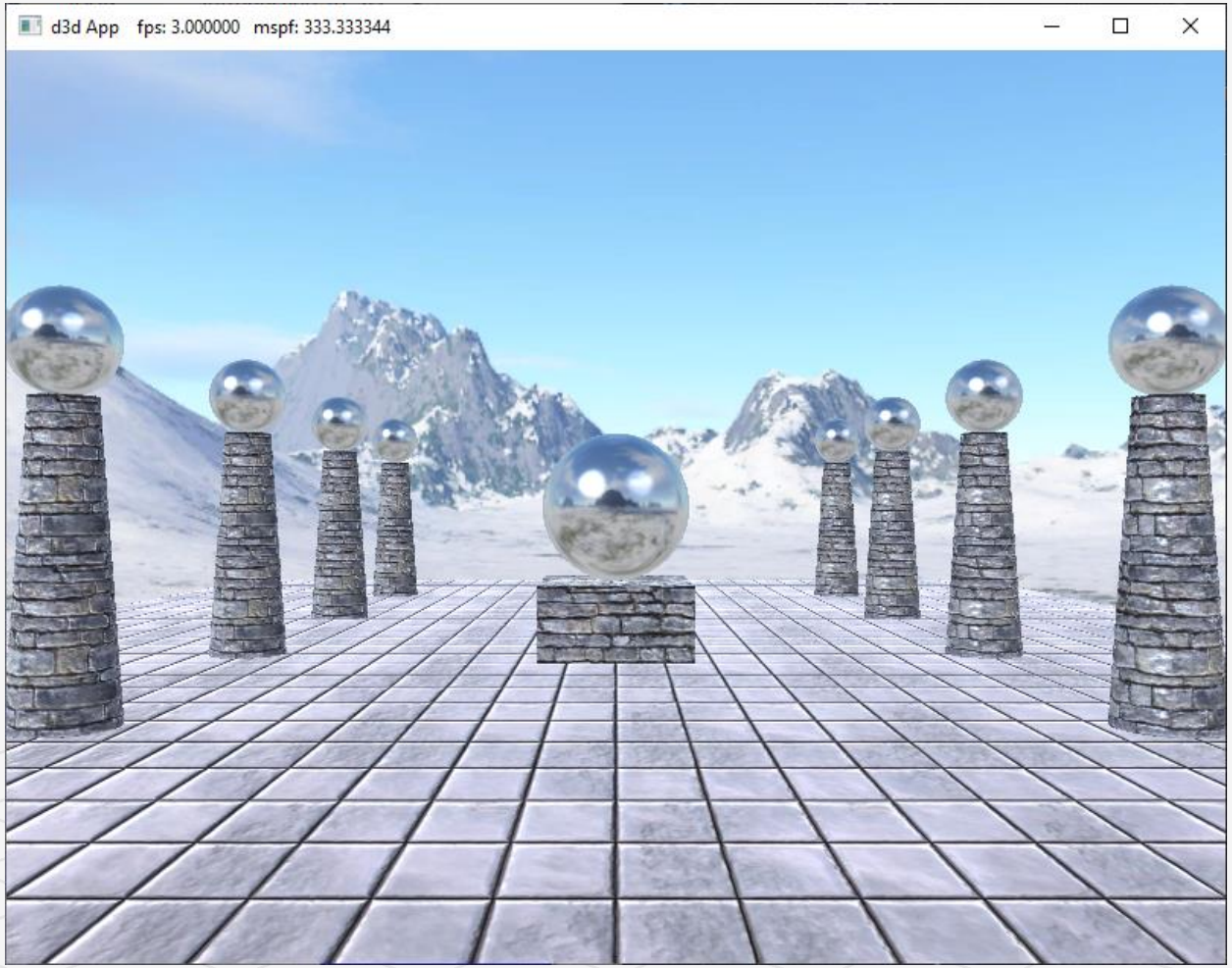
The alpha channel of the *tile_nmap.dds* image under “Textures” folders. The alpha channel denotes the shininess of the surface. White values indicate a shininess value of 1.0 and black values indicate a shininess value of 0.0. This gives us per-pixel control of the shininess material property.



SUMMARY

The strategy of normal mapping is to texture our polygons with normal maps. We then have per-pixel normals, which capture the fine details of a surface like bumps, scratches, and crevices. We then use these per-pixel normals from the normal map in our lighting calculations, instead of the interpolated vertex normal.

The coordinates of the normals in a normal map are relative to the texture space coordinate system. Consequently, to do lighting calculations, we need to transform the normal from the texture space to the world space so that the lights and normal are in the same coordinate system. The TBN-bases built at each vertex facilitates the transformation from texture space to world space.



SHADOW MAPPING

Objectives:

1. To discover the basic shadow mapping algorithm.

2. To learn how projective texturing works.

3. To find out about orthographic projections.

4. To understand shadow map aliasing problems and common strategies for fixing them.

RENDERING SCENE DEPTH

The shadow mapping algorithm relies on rendering the scene depth from the viewpoint of the light

This is essentially a variation of render-to-texture.

“rendering scene depth” = “building the depth buffer from the viewpoint of the light source”

After we have rendered the scene from the viewpoint of the light source, we will know the pixel fragments nearest to the light source: those fragments cannot be in shadow.

ShadowMap Utility

ShadowMap Utility helps us store the scene depth from the perspective of the light source.

It simply encapsulates a depth/stencil buffer, necessary views, and viewport.

A depth/stencil buffer used for shadow mapping is called a *shadow map*.

```
class ShadowMap
{
public:
ShadowMap(ID3D12Device* device,
UINT width, UINT height);

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

    UINT Width()const;
    UINT Height()const;
ID3D12Resource* Resource();
CD3DX12_GPU_DESCRIPTOR_HANDLE Srv()const;
CD3DX12_CPU_DESCRIPTOR_HANDLE Dsv()const;

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

void BuildDescriptors(
CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuSrv,
CD3DX12_GPU_DESCRIPTOR_HANDLE hGpuSrv,
CD3DX12_CPU_DESCRIPTOR_HANDLE hCpuDsv);

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_R24G8_TYPELESS;

CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuSrv;
CD3DX12_GPU_DESCRIPTOR_HANDLE mhGpuSrv;
CD3DX12_CPU_DESCRIPTOR_HANDLE mhCpuDsv;

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

ShadowMap::ShadowMap

The constructor creates the texture of the specified dimensions and viewport.

The resolution of the shadow map affects the quality of our shadows, but at the same time, a high resolution shadow map is more expensive to render into and requires more memory.

```
ShadowMap::ShadowMap(ID3D12Device* device, UINT width, UINT height)

{

md3dDevice = device;


mWidth = width;

mHeight = height;


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

mScissorRect = { 0, 0, (int)width, (int)height };


BuildResource();

}
```

Methods to access the shader resource and its views

1. The shadow mapping algorithm requires two render passes

a)we render the scene depth from the viewpoint of the light into the shadow map;
b)we render the scene as normal to the back buffer from our “player” camera,

2. Use the shadow map as a shader input to implement the shadowing algorithm.

We provide methods to access the shader resource and its views:

```
ID3D12Resource*  ShadowMap::Resource()
{
return mShadowMap.Get();
}

CD3DX12_GPU_DESCRIPTOR_HANDLE ShadowMap::Srv()const
{
return mhGpuSrv;
}

CD3DX12_CPU_DESCRIPTOR_HANDLE ShadowMap::Dsv()const
{
return mhCpuDsv;
}
```

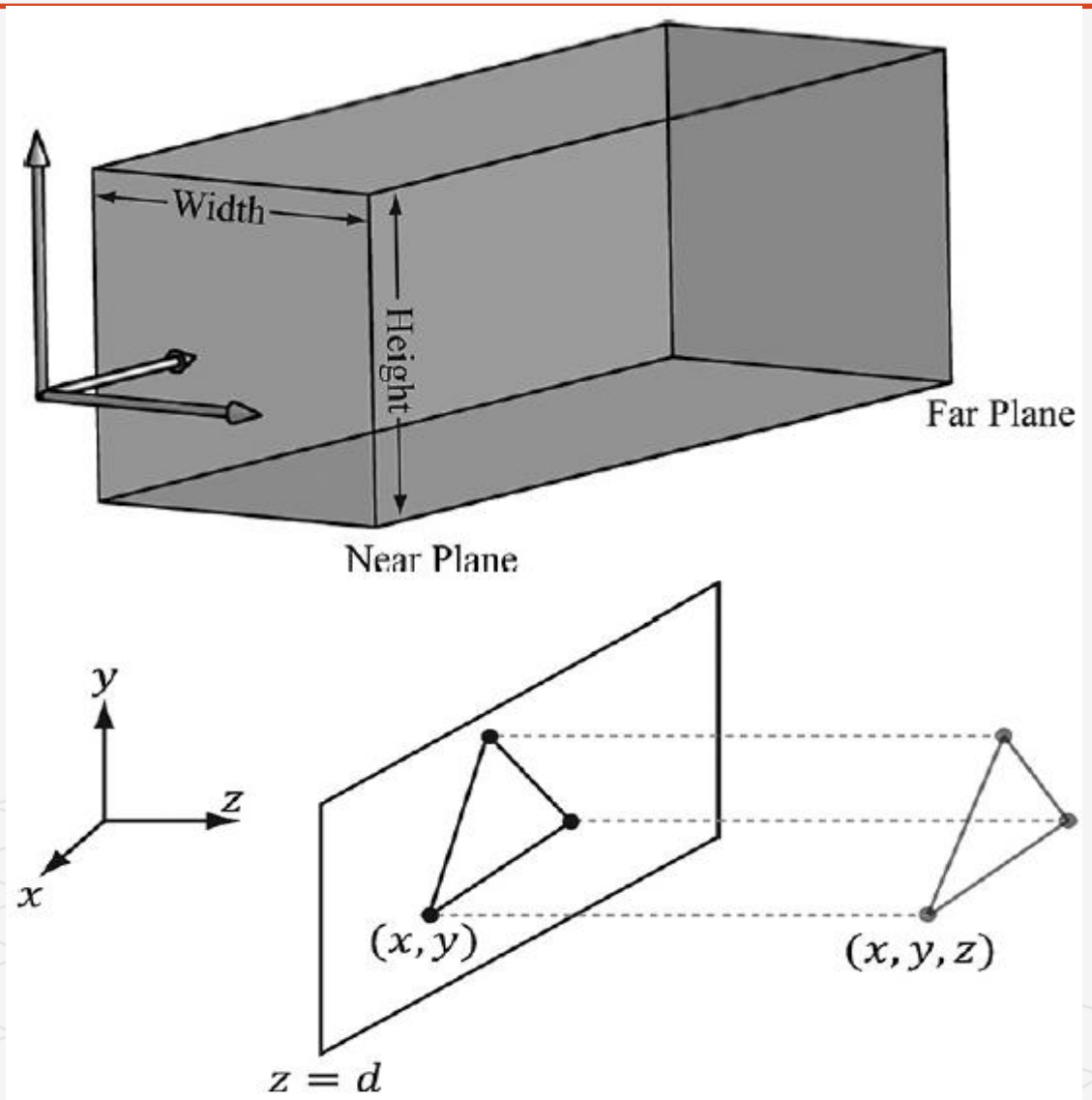
ORTHOGRAPHIC PROJECTIONS

Orthographic projections will enable us to model shadows that parallel lights generate.

With an orthographic projection, the viewing volume is a box axis-aligned with the view space with width *w*, height *h*, near plane *n* and face plane *f* that looks down the positive *z*-axis of view space.

With an orthographic projection, the lines of projection are parallel to the view space *z*-axis.

Note that the 2D projection of a vertex (*x*, *y*, *z*) is just (*x*, *y*).



orthographic projection matrix

To transform the view volume from view space to NDC space, we need to rescale and shift to map the view space view volume

$$\left[-\frac{w}{2},\frac{w}{2}\right]\times\left[-\frac{h}{2},\frac{h}{2}\right]\times[n,f]$$

to the NDC space view volume [−1, 1] × [-1, 1] × [0, 1]. For first two coordinates:

$$\frac{2}{w}\cdot\left[-\frac{w}{2},\frac{w}{2}\right]=[-1,1]$$

$$\frac{2}{h}\cdot\left[-\frac{h}{2},\frac{h}{2}\right]=[-1,1]$$

For the third coordinate, we need to map [*n*, *f*] → [0, 1].

$$[x',y',z',1]=[x,y,z,1]\begin{bmatrix}\frac{2}{w} & 0 & 0 & 0\\ 0 & \frac{2}{h} & 0 & 0\\ 0 & 0 & \frac{1}{f-n} & 0\\ 0 & 0 & \frac{n}{n-f} & 1\end{bmatrix}$$

PROJECTIVE TEXTURE COORDINATES

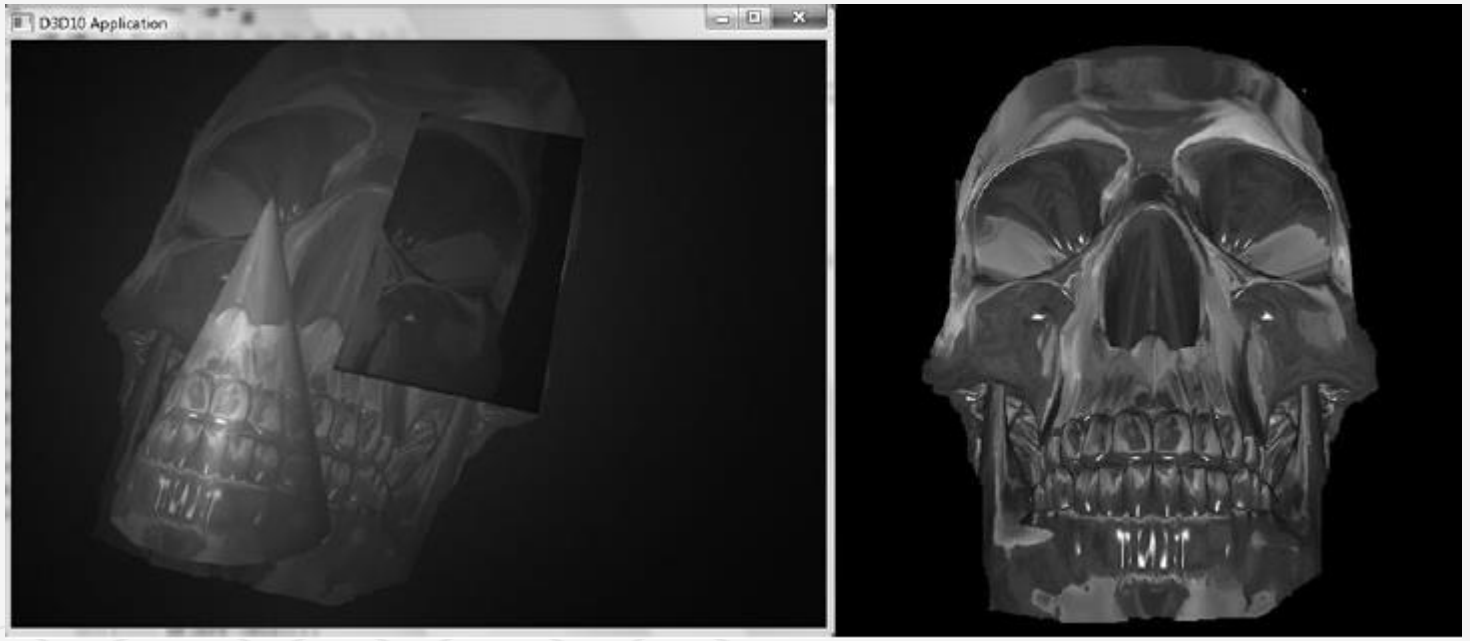
Projective texturing allows us to project a texture onto arbitrary geometry, much like a slide projector.

The skull texture (right) is projected onto the scene geometry (left).

Projective texturing is also used as an intermediate step for shadow mapping.

The key to projective texturing is to generate texture coordinates for each pixel in such a way that the applied texture looks like it has been projected onto the geometry.

We will call such generated texture coordinates *projective texture coordinates*.



Generating projective texture coordinates

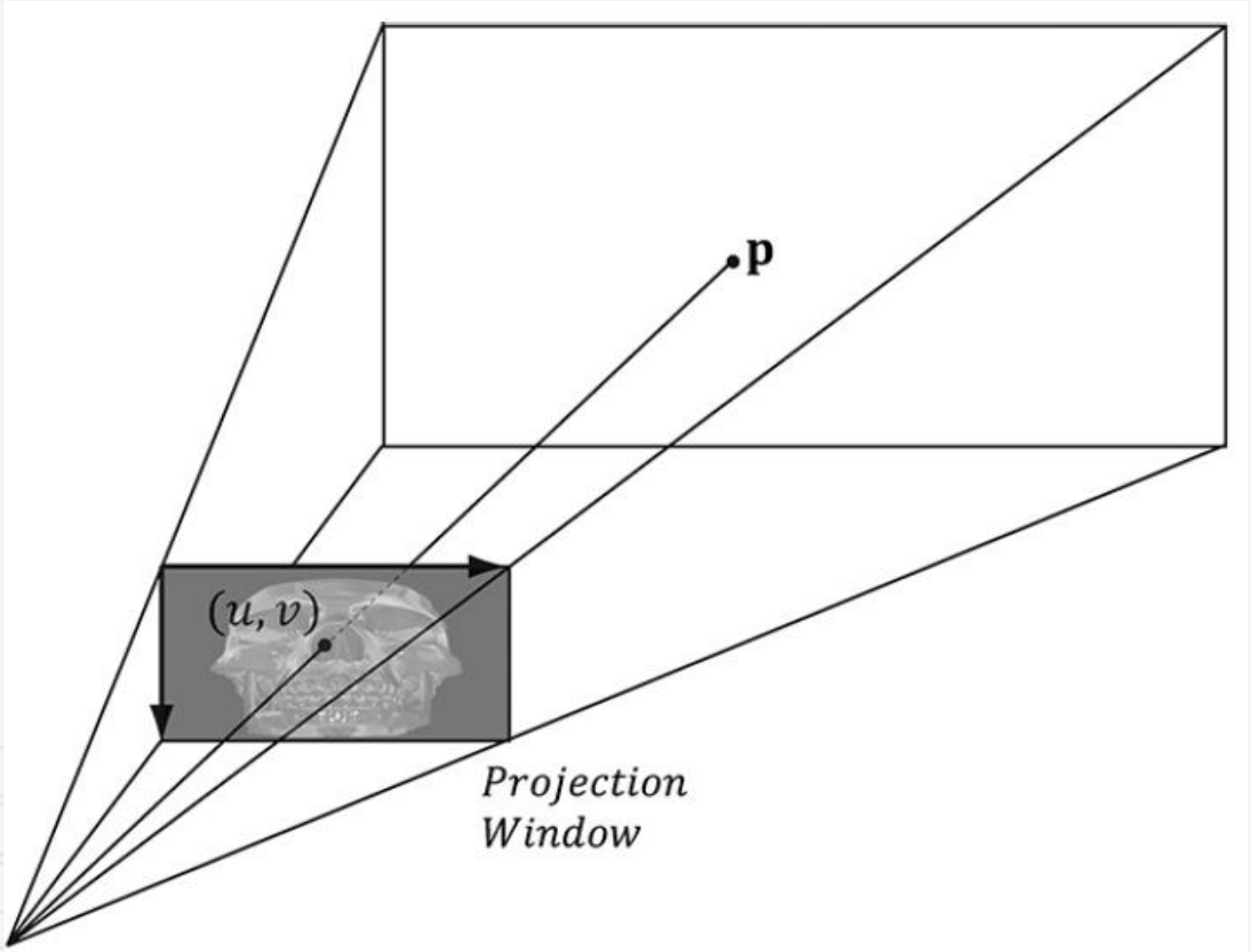
The texture coordinates (*u*, *v*) identify the texel that should be projected onto the 3D point **p**.

The texel identified by the coordinates (*u*, *v*) relative to the texture space on the projection window is projected onto the point **p** by following the line of sight from the light origin to the point **p**.

So the strategy of generating projective texture coordinates is as follows:

Step 1. Project the point **p** onto the light's projection window and transform the coordinates to NDC space.

Step 2. Transform the projected coordinates from NDC space to texture space, thereby effectively turning them into texture coordinates.



Generating projective texture coordinates

Step 1

By thinking of the light projector as a camera, we define a view matrix **V** and projection matrix **P** for the light projector.

These matrices essentially define the position, orientation, and frustum of the light projector in the world.

The matrix **V** transforms coordinates from world space to the coordinate system of the light projector.

Once the coordinates are relative to the light coordinate system, the projection matrix, along with the homogeneous divide, are used to project the vertices onto the projection plane of the light.

After the homogeneous divide, the coordinates are in NDC space.

Step 2

Transform from NDC space to texture space via the following change of coordinate transformation:

$$\boldsymbol{u = 0.5x + 0.5}$$

$$\boldsymbol{v = -0.5y + 0.5}$$

Here, u, $v \in [0, 1]$ provided x, $y \in [-1, 1]$. We scale the y-coordinate by a negative to invert the axis because the positive y-axis in NDC coordinates goes in the direction opposite to the positive v-axis in texture coordinates.

The texture space transformations can be written in terms of matrices. The below matrix **T** for "texture matrix" that transforms from NDC space to texture space. We can form the composite transform **VPT** that takes us from world space directly to texture space. After we multiply by this transform, we still need to do the perspective divide to complete the transformation

$$\begin{bmatrix} x, & y, & 0, & 1 \end{bmatrix} \begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u, & v, & 0, & 1 \end{bmatrix}$$

Code Implementation

```
struct VertexOut
{
float4 PosH : SV_POSITION;
float3 PosW : POSITION;
float3 TangentW : TANGENT;
float3 NormalW : NORMAL;
float2 Tex : TEXCOORD0;
float4 ProjTex : TEXCOORD1;
};
VertexOut VS(VertexIn vin)
{
VertexOut vout;
[…]
// Transform to light’s projective space.
vout.ProjTex = mul(float4(vIn.posL, 1.0f),
gLightWorldViewProjTexture);
[…]
return vout;
}
```

```
float4 PS(VertexOut pin) : SV_Target
{
// Complete projection by doing division by w.
pin.ProjTex.xyz /= pin.ProjTex.w;
// Depth in NDC space.
float depth = pin.ProjTex.z;
// Sample the texture using the projective texcoords.
float4 c = gTextureMap.Sample(sampler,
pin.ProjTex.xy);
[…]
}
```

Points Outside the Frustum

- In the rendering pipeline, geometry outside the frustum is clipped.
- When we generate projective texture coordinates by projecting the geometry from the point of view of the light projector, no clipping is done.
- Geometry outside the projector's frustum receives projective texture coordinates outside the [0, 1] range.
- Generally, we do not want to texture any geometry outside the projector's frustum because it does not make sense. Such geometry receives no light from the projector.
- One solution is to associate a spotlight with the projector so that anything outside the spotlight's field of view cone is not lit (i.e., the surface receives no projected light).

Orthographic Projections

The texture is projected in the direction of the *z*-axis of the light through a box.

With an orthographic projection, the spotlight strategy used to handle points outside the projector's volume does not work. This is because a spotlight cone approximates the volume of a frustum to some degree, but it does not approximate a box.

We can still use texture address modes to handle points outside the projector's volume. This is because an orthographic projection still generates NDC coordinates and a point (*x*, *y*, *z*) is inside the volume if and only if: $-1 \leq x \leq 1, -1 \leq y \leq 1, 0 \leq z \leq 1$

With an orthographic projection, we do not need to do the divide by *w*; that is, we do not need the line. Because, after an orthographic projection, the coordinates are already in NDC space.

```
// Complete projection by doing division by w.

pin.ProjTex.xyz /= pin.ProjTex.w;
```

SHADOW MAPPING

1. The idea of the shadow mapping algorithm is to render-to-texture the scene depth from the viewpoint of the light into a depth buffer called a *shadow map*.

2. The shadow map will contain the depth values of all the visible pixels from the perspective of the light.

3. Pixels occluded by other pixels will not be in the shadow map because they will fail the depth test and either be overwritten or never written.

4. To render the scene from the viewpoint of the light, we need to define a light view matrix that transforms coordinates from world space to the space of the light.

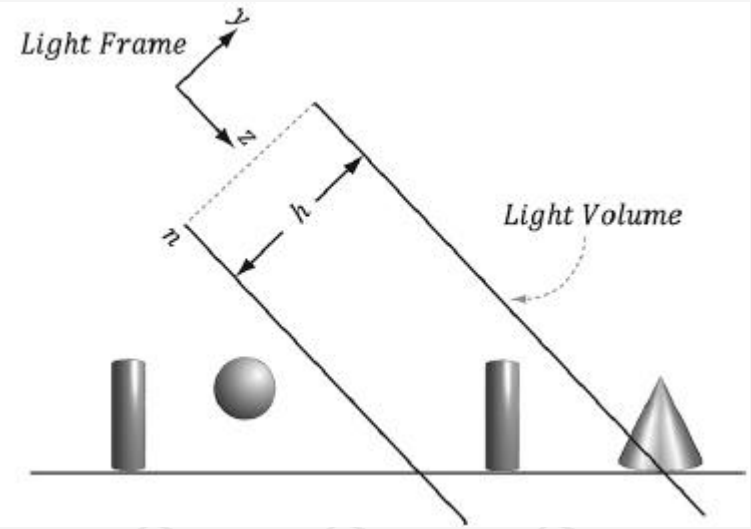
5. A light projection matrix describes the volume that light emits through in the world.

6. This volume can be either a frustum volume (perspective projection) or box volume (orthographic projection).

7. A frustum light volume can be used to model spotlights by embedding the spotlight cone inside the frustum.

8. A box light volume can be used to model parallel lights. The following figure show how parallel lights

only strike a subset of the scene.



SHADOW MAPPING

Once we have built the shadow map, we render the scene as normal from the perspective of the "player" camera.

For each pixel p rendered, we also compute its depth from the light source, which we denote by $d(p)$.

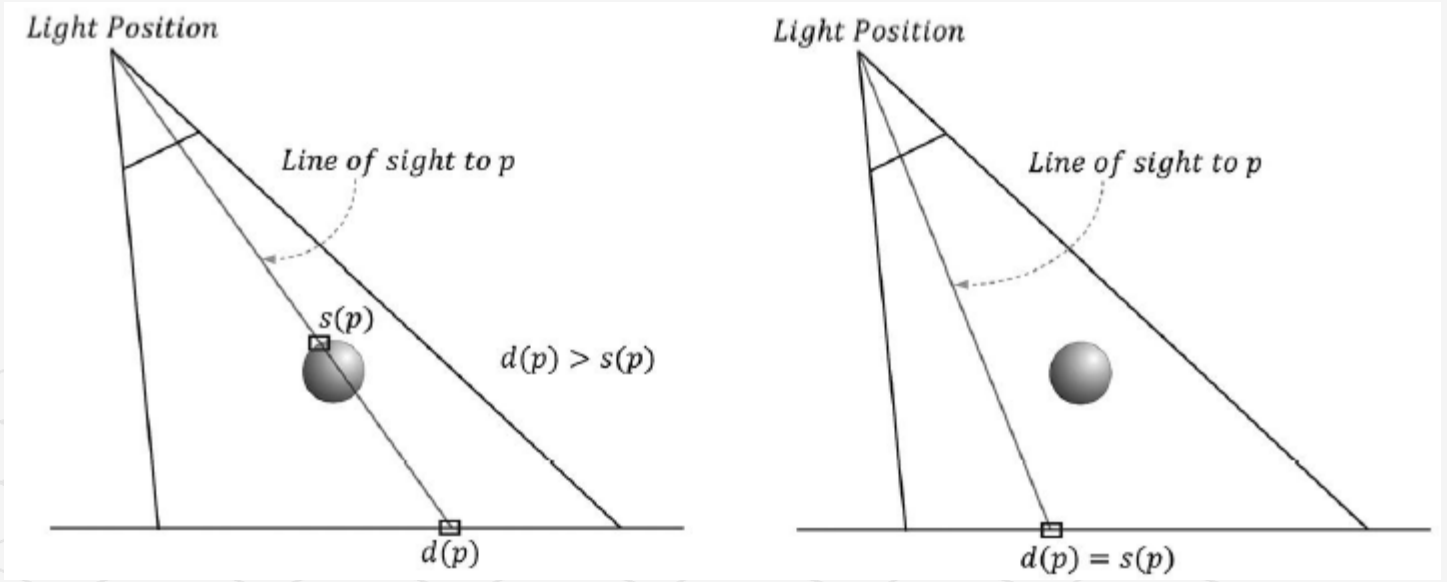
In addition, using projective texturing, we sample the shadow map along the line of sight from the light source to the pixel p to get the depth value $s(p)$ stored in the shadow map;

This value is the depth of the pixel closest to the light along the line of sight from the position of the light to p.

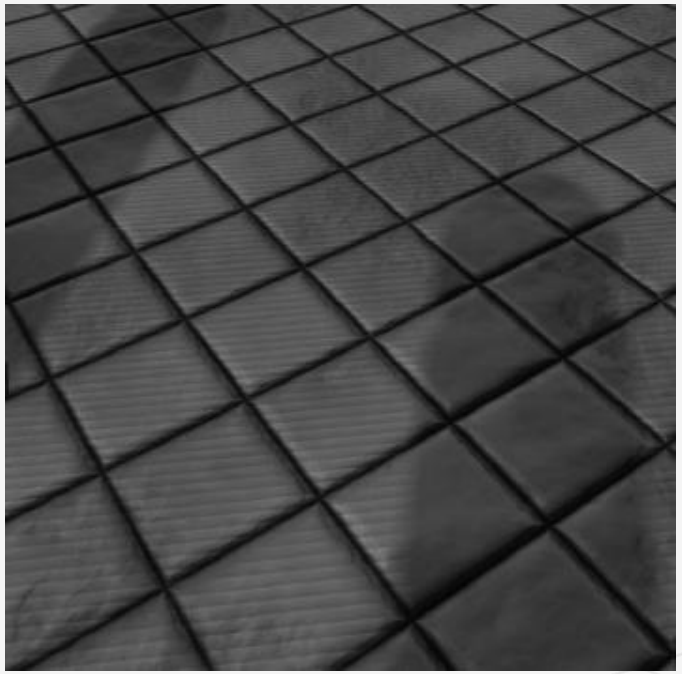
Note that a pixel p is in shadow if and only if $d(p) > s(p)$. Hence a pixel is not in shadow if and only if $d(p) \leq s(p)$.

On the left, the depth of the pixel p from the light is $d(p)$. However, the depth of the pixel nearest to the light along the same line of sight has depth $s(p)$, and $d(p) > s(p)$. We conclude, therefore, that there is an object in front of p from the perspective of the light and so p is in shadow.

On the right, the depth of the pixel p from the light is $d(p)$ and it also happens to be the pixel nearest to the light along the line of sight, that is, $s(p) = d(p)$, so we conclude p is not in shadow.



Biasing and Aliasing



The shadow map stores the depth of the nearest visible pixels with respect to its associated light source.

Each shadow map texel corresponds to an area of the scene which is just a discrete sampling of the scene depth from the light perspective.

Because the shadow map only has some finite resolution, this causes aliasing issues known as *shadow acne*.

Shadow Acne

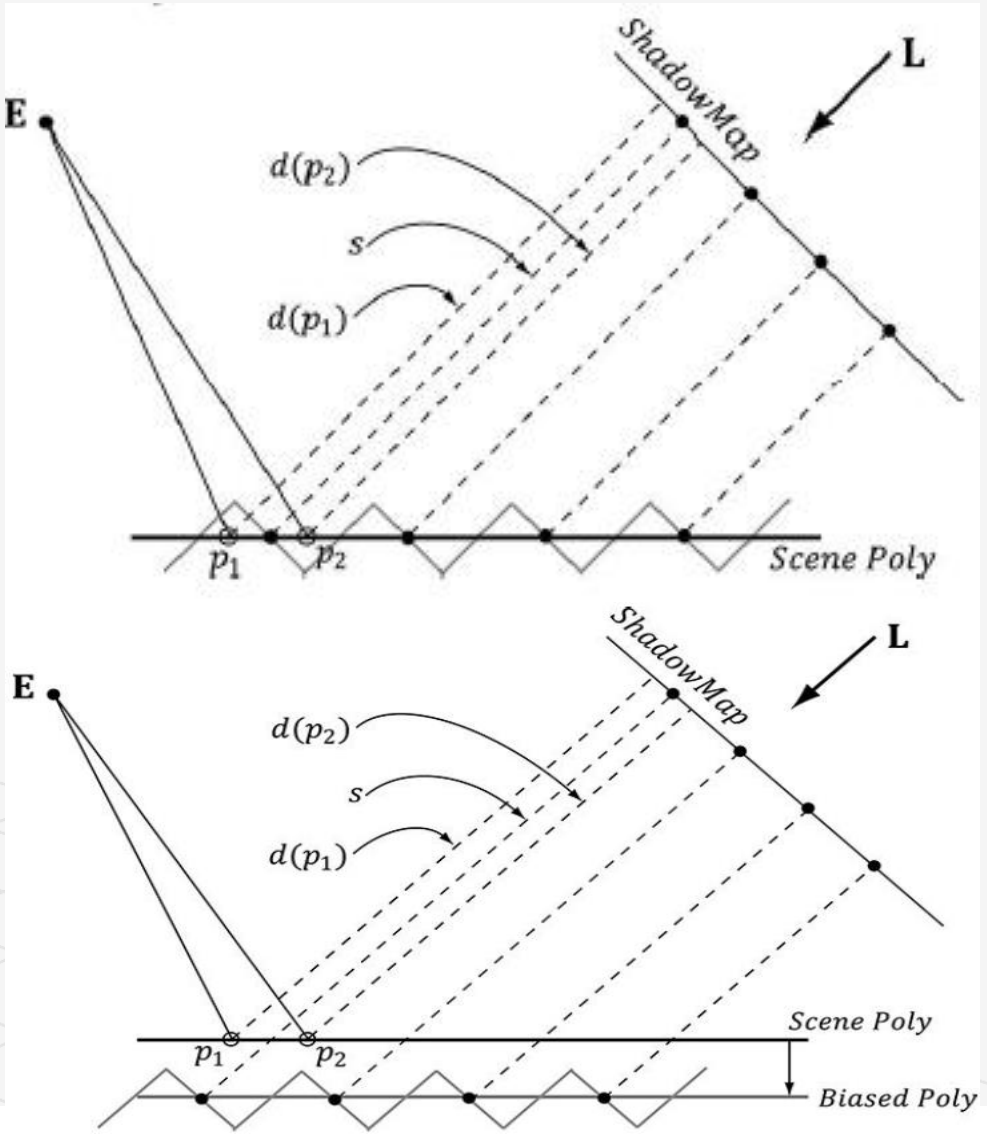
The eye **E** sees two points on the scene $p1$ and $p2$ that correspond to different screen pixels.

However, from the viewpoint of the light, both points are covered by the same shadow map texel (that is, $s(p1) = s(p2) = s$).

When we do the shadow map test, we have $d(p1) > s$ and $d(p2) \leq s$.

Therefore, $p1$ will be colored as if it were in shadow, and $p2$ will be colored as if it were not in shadow. This causes the shadow acne.

A simple solution is to apply a constant bias to offset the shadow map depth. By biasing the depth values in the shadow map, no false shadowing occurs. We have that $d(p1) \leq s$ and $d(p2) \leq s$. Finding the right depth bias is usually done by experimentation.



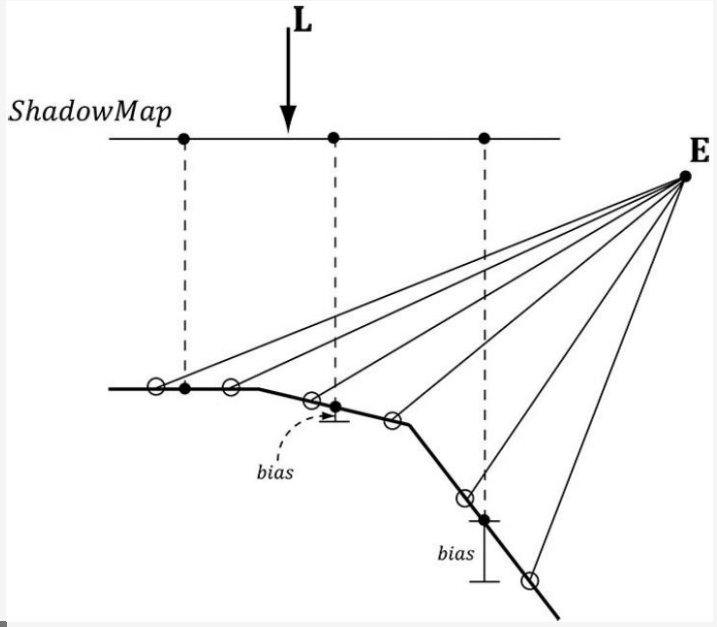
Peter-Panning

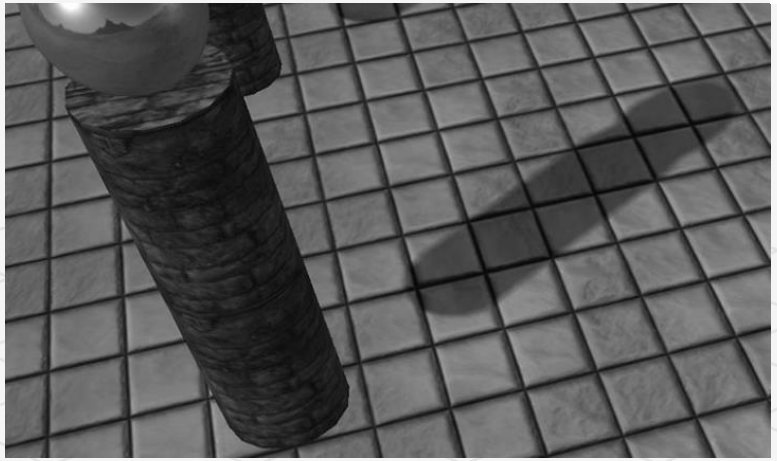
Too much biasing results in an artifact called *peter-panning*, where the shadow appears to become detached from the object.

The term *Peter Panning* derives its name from a children's book character whose shadow became detached and who could fly.

A fixed bias does not work for all geometry. In particular, Figure shows that triangles with large slopes (with respect to the light source) need a larger bias.

It is tempting to choose a large enough depth bias to handle all slopes. However, as Figure showed, this leads to peter-panning.





slope-scaled-bias rasterization state

What we want is a way to measure the polygon slope with respect to the light source, and apply more bias for larger sloped polygons.

Fortunately, graphics hardware has intrinsic support for this via *slope-scaled-bias* rasterization state properties:

1. DepthBias: A fixed bias to apply.

2. DepthBiasClamp: A maximum depth bias allowed. This allows us to set a bound on the depth bias, for we can imagine that for very steep slopes, the bias slope-scaled-bias would be too much and cause peter-panning artifacts.

3. SlopeScaledDepthBias: A scale factor to control how much to bias based on the polygon slope.

```
typedef struct D3D12_RASTERIZER_DESC {
  D3D12_FILL_MODE                 FillMode;
  D3D12_CULL_MODE                 CullMode;
  BOOL                            FrontCounterClockwise;
  INT                             DepthBias;
  FLOAT                           DepthBiasClamp;
  FLOAT                           SlopeScaledDepthBias;
  BOOL                            DepthClipEnable;
  BOOL                            MultisampleEnable;
  BOOL                            AntialiasedLineEnable;
  UINT                            ForcedSampleCount;
  D3D12_CONSERVATIVE_RASTERIZATION_MODE ConservativeRaster;
} D3D12_RASTERIZER_DESC;
```

PSO for shadow map pass

```
// [From MSDN]
// If the depth buffer currently bound to the outputmerger stage has a UNORM format or no depth buffer is bound the bias value
// is calculated like this:
//
// Bias = (float)DepthBias * r + SlopeScaledDepthBias* MaxDepthSlope;
//
// where r is the minimum representable value > 0 in the
// depth-buffer format converted to float32.
//
// For a 24-bit depth buffer, r = 1 / 2^24.
//
// Example: DepthBias = 100000 ==> Actual DepthBias = 100000 / 2 ^ 24 = .006
// These values are highly scene dependent, and you will need
// to experiment with these values for your scene to find the
// best values.

D3D12_GRAPHICS_PIPELINE_STATE_DESC smapPsoDesc = opaquePsoDesc;
    smapPsoDesc.RasterizerState.DepthBias = 100000;
    smapPsoDesc.RasterizerState.DepthBiasClamp = 0.0f;
    smapPsoDesc.RasterizerState.SlopeScaledDepthBias = 1.0f;
    smapPsoDesc.pRootSignature = mRootSignature.Get();
```

PCF (*Percentage Closer Filtering*)

The projective texture coordinates (*u*, *v*) used to sample the shadow map generally will not coincide with a texel in the shadow map.

Usually, it will be between four texels. With color texturing, this is solved with bilinear interpolation.

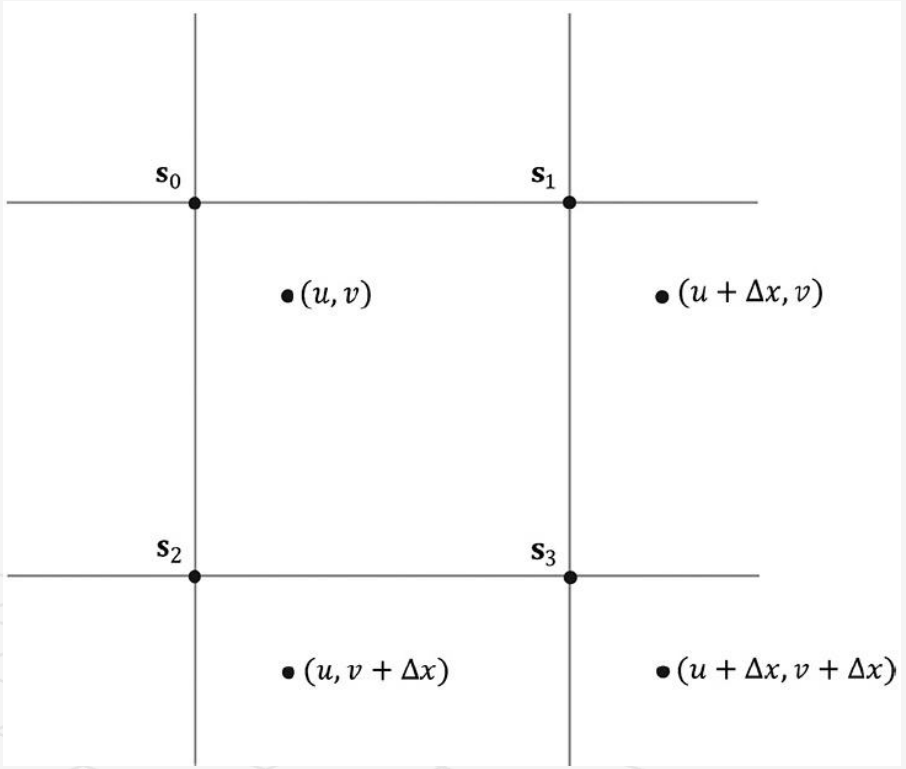
Interpolation depth values can lead to incorrect results about a pixel being flagged in shadow.

percentage closer filtering (PCF) uses point filtering (MIN_MAG_MIP_POINT) and sample the texture with coordinates:

(*u*, *v*), (*u* + Δ*x*, *v*), (*u*, *v* + Δ*x*), (*u* + Δ*x*, *v* + Δ*x*),

where Δ*x* = 1/SHADOW_MAP_SIZE.

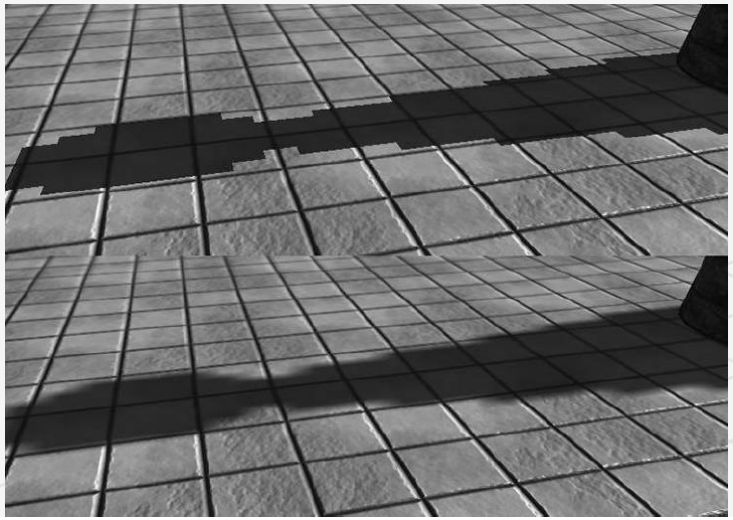
Since we are using point sampling, these four points will hit the nearest four texels **s**0, **s**1, **s**2, and **s**3, respectively, surrounding (*u*, *v*),



The shadow map test

We then do the shadow map test for each of these sampled depths and bilinearly interpolate the shadow map results:

In this way, it is not an all-or-nothing situation; a pixel can be partially in shadow. For example, if two of the samples are in shadow and two are not in shadow, then the pixel is 50% in shadow. This creates a smoother transition from shadowed pixels to non-shadows pixels.



```
static const float SMAP_SIZE = 2048.0f;
static const float SMAP_DX = 1.0f / SMAP_SIZE;
…
// Sample shadow map to get nearest depth to light.
float s0 = gShadowMap.Sample(gShadowSam,projTexC.xy).r;
float s1 = gShadowMap.Sample(gShadowSam,projTexC.xy + float2(SMAP_DX, 0)).r;
float s2 = gShadowMap.Sample(gShadowSam,projTexC.xy + float2(0, SMAP_DX)).r;
float s3 = gShadowMap.Sample(gShadowSam,projTexC.xy + float2(SMAP_DX, SMAP_DX)).r;
// Is the pixel depth <= shadow map value?
float result0 = depth <= s0;
float result1 = depth <= s1;
float result2 = depth <= s2;
float result3 = depth <= s3;
// Transform to texel space.
float2 texelPos = SMAP_SIZE * projTexC.xy;
// Determine the interpolation amounts.
float2 t = frac(texelPos);
// Interpolate results.
return lerp(lerp(result0, result1, t.x),lerp(result2, result3, t.x), t.y);
```

SampleCmpLevelZero

The main disadvantage of PCF filtering is that it requires four texture samples.

Sampling textures is one of the more expensive operations on a modern GPU because memory bandwidth and memory latency have not improved as much as the raw computational power of GPUs.

Fortunately, Direct3D 11+ graphics hardware has built in support for PCF via the SampleCmpLevelZero method.

The LevelZero part of the method name means that it only looks at the top mipmap level.

This method does not use a typical sampler object, but instead uses a so-called *comparison sampler*.

```
Texture2D gShadowMap : register(t1);

SamplerComparisonState gsamShadow : register(s6);

// Complete projection by doing division by w.

shadowPosH.xyz /= shadowPosH.w;

// Depth in NDC space.

float depth = shadowPosH.z;

// Automatically does a 4-tap PCF.

gShadowMap.SampleCmpLevelZero(gsamShadow, shadowPosH.xy, depth).r;
```

CD3DX12_STATIC_SAMPLER_DESC

For PCF, you need to use the filter D3D12_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT and set the comparison function to LESS_EQUAL.

```
const CD3DX12_STATIC_SAMPLER_DESC shadow(
        6, // shaderRegister
        D3D12_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINT, // filter
        D3D12_TEXTURE_ADDRESS_MODE_BORDER,  // addressU
        D3D12_TEXTURE_ADDRESS_MODE_BORDER,  // addressV
        D3D12_TEXTURE_ADDRESS_MODE_BORDER,  // addressW
        0.0f,                               // mipLODBias
        16,                                 // maxAnisotropy
        D3D12_COMPARISON_FUNC_LESS_EQUAL,
        D3D12_STATIC_BORDER_COLOR_OPAQUE_BLACK);
```

Building the Shadow Map

The first step in shadow mapping is building the shadow map. To do this, we create a ShadowMap instance:

```
mShadowMap = std::make_unique<ShadowMap>(
md3dDevice.Get(), 2048, 2048);
```

We then define a light view matrix and projection matrix (representing the light frame and view volume).

The light view matrix is derived from the primary light source, and the light view volume is computed to fit the bounding sphere of the entire scene.

```
DirectX::BoundingSphere mSceneBounds;
```

```
ShadowMapApp::ShadowMapApp(HINSTANCE hInstance)
    : D3DApp(hInstance)
{
// Estimate the scene bounding sphere manually since we know how the scene was constructed.
// The grid is the "widest object" with a width of 20 and depth of 30.0f, and centered at
// the world space origin.  In general, you need to loop over every world space vertex
// position and compute the bounding sphere.
    mSceneBounds.Center = XMFLOAT3(0.0f, 0.0f, 0.0f);
    mSceneBounds.Radius = sqrtf(10.0f*10.0f + 15.0f*15.0f);
}
```

ShadowMapApp::BuildPSOs

Note that we set a null render target, which essentially disables color writes.

This is because when we render the scene to the shadow map, all we care about is the depth values of the scene relative to the light source.

Graphics cards are optimized for only drawing depth; a depth only render pass is significantly faster than drawing color anddepth.

The active pipeline state object must also specify a render target count of 0:

```
smapPsoDesc.VS =
    {
        reinterpret_cast<BYTE*>(mShaders["shadowVS"]->GetBufferPointer()),
        mShaders["shadowVS"]->GetBufferSize()
    };
    smapPsoDesc.PS =
    {
        reinterpret_cast<BYTE*>(mShaders["shadowOpaquePS"]->GetBufferPointer()),
        mShaders["shadowOpaquePS"]->GetBufferSize()
    };

    // Shadow map pass does not have a render target.
    smapPsoDesc.RTVFormats[0] = DXGI_FORMAT_UNKNOWN;
    smapPsoDesc.NumRenderTargets = 0;
    ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&smapPsoDesc,
IID_PPV_ARGS(&mPSOs["shadow_opaque"])));
```

Shadow.hlsl

The shader programs we use for rendering the scene from the perspective of the light is quite simple because we are only building the shadow map, so we do not need to do any complicated pixel shader work.

Notice that the pixel shader does not return a value because we only need to output depth values.

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

struct VertexOut
{
float4 PosH    : SV_POSITION;
float2 TexC    : TEXCOORD;
};
```

```
VertexOut VS(VertexIn vin)
{
VertexOut vout = (VertexOut)0.0f;

MaterialData matData = gMaterialData[gMaterialIndex];

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

void PS(VertexOut pin)
{
// Fetch the material data.
MaterialData matData = gMaterialData[gMaterialIndex];
float4 diffuseAlbedo = matData.DiffuseAlbedo;
uint diffuseMapIndex = matData.DiffuseMapIndex;

// Dynamically look up the texture in the array.
diffuseAlbedo *= gTextureMaps[diffuseMapIndex].Sample(gsamAnisotropicWrap,
pin.TexC);

#ifdef ALPHA_TEST
clip(diffuseAlbedo.a - 0.1f);
#endif
}
```

Pixel shader

The pixel shader is solely used to clip pixel fragments with zero or low alpha values, which we assume indicate complete transparency.

For example, consider the tree leaf texture in Figure. here, we only want to draw the pixels with white alpha values to the shadow map. To facilitate this, we provide two techniques:

one that does the alpha clip operation, and one that does not.

If the alpha clip does not need to be done, then we can bind a null pixel shader, which would be even faster than binding a pixel shader that only samples a texture and performs a clip operation.

The Shadow Factor

The shadow factor is a scalar in the range 0 to 1.

A value of 0 indicates a point is in shadow, and a value of 1 indicates a point is not in shadow.

With PCF, a point can also be partially in shadow, in which case the shadow factor will be between 0 and 1.

The CalcShadowFactor implementation is in *Common.hlsl*.

```
float CalcShadowFactor(float4 shadowPosH)
{
    // Complete projection by doing division by w.
    shadowPosH.xyz /= shadowPosH.w;

    // Depth in NDC space.
    float depth = shadowPosH.z;

    uint width, height, numMips;
    gShadowMap.GetDimensions(0, width, height, numMips);

    // Texel size.
    float dx = 1.0f / (float)width;

    float percentLit = 0.0f;
    const float2 offsets[9] =
    {
        float2(-dx,  -dx), float2(0.0f,  -dx), float2(dx,  -dx),
        float2(-dx, 0.0f), float2(0.0f, 0.0f), float2(dx, 0.0f),
        float2(-dx,  +dx), float2(0.0f,  +dx), float2(dx,  +dx)
    };

    [unroll]
    for(int i = 0; i < 9; ++i)
    {
        percentLit += gShadowMap.SampleCmpLevelZero(gsamShadow,
            shadowPosH.xy + offsets[i], depth).r;
    }

    return percentLit / 9.0f;
}
```

The shadow factor

The shadow factor does not affect ambient light since that is indirect light, and it also does not affect reflective light coming from the environment map.

```
// Only the first light casts a shadow.
    float3 shadowFactor = float3(1.0f, 1.0f, 1.0f);
    shadowFactor[0] = CalcShadowFactor(pin.ShadowPosH);

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

    float4 litColor = ambient + directLight;
```

ComputeLighting

In our model, the shadow factor will be multiplied against the direct lighting (diffuse and specular) terms:

```
float4 ComputeLighting(Light gLights[MaxLights], Material mat,
                       float3 pos, float3 normal, float3 toEye,
                       float3 shadowFactor)
{
    float3 result = 0.0f;

    int i = 0;

#if (NUM_DIR_LIGHTS > 0)
    for(i = 0; i < NUM_DIR_LIGHTS; ++i)
    {
        result += shadowFactor[i] * ComputeDirectionalLight(gLights[i], mat, normal, toEye);
    }
#endif

#if (NUM_POINT_LIGHTS > 0)
    for(i = NUM_DIR_LIGHTS; i < NUM_DIR_LIGHTS+NUM_POINT_LIGHTS; ++i)
    {
        result += ComputePointLight(gLights[i], mat, pos, normal, toEye);
    }
#endif

#if (NUM_SPOT_LIGHTS > 0)
    for(i = NUM_DIR_LIGHTS + NUM_POINT_LIGHTS; i < NUM_DIR_LIGHTS + NUM_POINT_LIGHTS +
NUM_SPOT_LIGHTS; ++i)
    {
        result += ComputeSpotLight(gLights[i], mat, pos, normal, toEye);
    }
#endif

    return float4(result, 0.0f);
}
```

The Shadow Map Test

After we have built the shadow map by rendering the scene from the perspective of the light, we can sample the shadow map in our main rendering pass to determine if a pixel is in shadow or not.

$d(p)$ is found by transforming the point to the NDC space of the light;

the z-coordinate gives the normalized depth value of the point from the light source.

$s(p)$ is found by projecting the shadow map onto the scene through the light's view volume using projective texturing.

```
cbuffer cbPass : register(b1)
{
    float4x4 gView;
    float4x4 gInvView;
    float4x4 gProj;
    float4x4 gInvProj;
    float4x4 gViewProj;
    float4x4 gInvViewProj;

// The gShadowTransform matrix is stored as a per-pass constant.
    float4x4 gShadowTransform;
…

VertexOut VS(VertexIn vin)
{
….
// Generate projective tex-coords to project shadow map onto scene.
vout.ShadowPosH = mul(posW, gShadowTransform);

float4 PS(VertexOut pin) : SV_Target
{
// Do the shadow map test in pixel shader.
// Only the first light casts a shadow.
    float3 shadowFactor = float3(1.0f, 1.0f, 1.0f);
    shadowFactor[0] = CalcShadowFactor(pin.ShadowPosH);
```

Rendering the Shadow Map

For this demo, we also render the shadow map onto a quad that occupies the lower right corner of the screen.

This allows us to see what the shadow map looks like for each frame.

Recall that the shadow map is just a depth buffer texture and we can create an SRV to it so that it can be sampled in a shader program.

The shadow map is rendered as a grayscale image since it stores a one-dimensional value at each pixel (a depth value).