Game Engine Development II

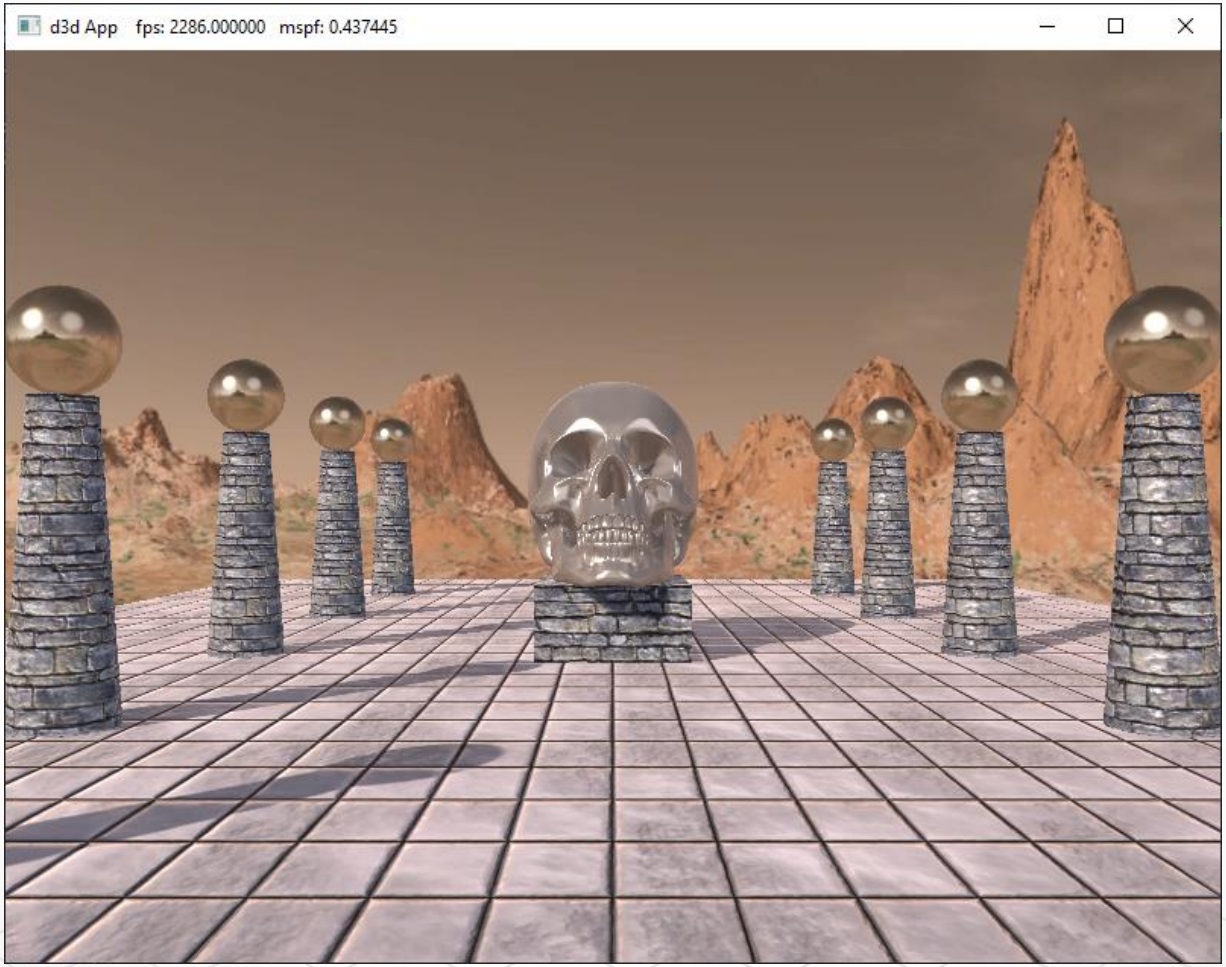
Week5

Hooman Salamat

Objectives

- To discover the basic shadow mapping algorithm.
- To learn how projective texturing works.
- To find out about orthographic projections.
- To understand shadow map aliasing problems and common strategies for fixing them.

SHADOW MAPPING



RENDERING SCENE DEPTH

- Shadows indicate to the observer where light originates and helps convey the relative locations of objects in a scene.
- 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.

```
//The first step in shadow mapping is to build the shadow map. To do this, we create a ShadowMap instance

//Try with low resolution 512x512 to experiment perspective aliasing

    mShadowMap = std::make_unique<ShadowMap>(

        md3dDevice.Get(), 2048, 2048);
```

```
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

The shadow mapping algorithm requires two render passes:

1) Render the scene depth from the viewpoint of the light into the shadow map;

2) Render the scene as normal to the back buffer from our "player" camera,

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

We provide these 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;
}
```

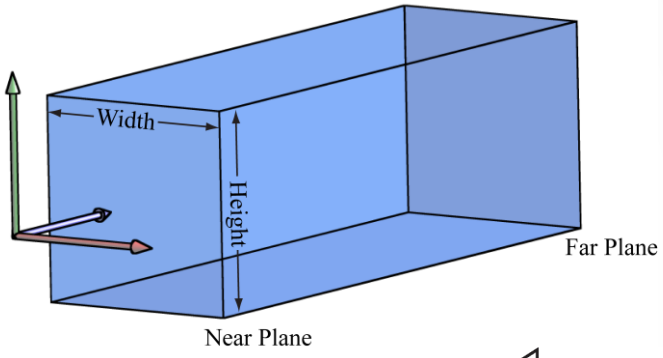
ORTHOGRAPHIC PROJECTIONS

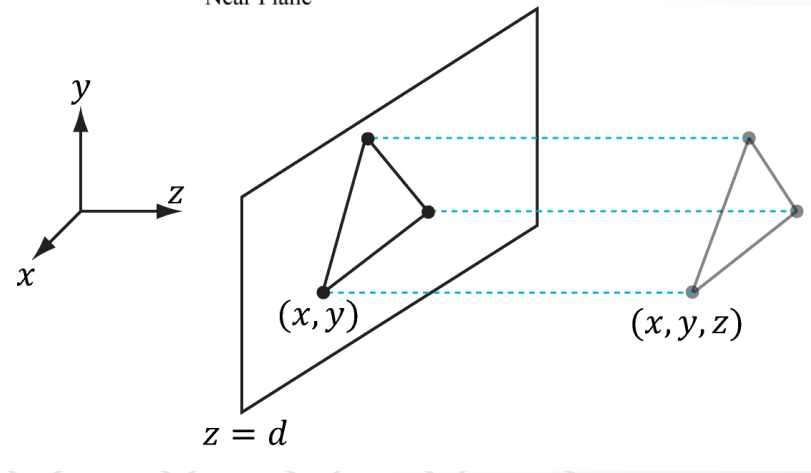
Orthographic projections will enable us to model shadows that parallel lights generate.

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

As with perspective projection, we want to maintain relative depth information, and we want normalized device coordinates.

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:

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

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

ed to map [*n*, *f*] → [0, 1]. We have the conditions *g*(*n*) = 0 and *g*(*f*) = 1, which allow us to solve for *a* and *b*:

$$an+b=0 \text{ and } af+b=1$$

The 4 × 4 matrix in the equation is the *orthographic projection matrix*.

Recall that with the perspective projection transform, we had to split it into two parts:

A linear part described by the projection matrix, and a nonlinear part described by the

divide by *w*. In contrast, the orthographic projection transformation is completely linear—

there is no divide by *w*. Multiplying by the orthographic projection matrix takes us directly

into NDC coordinates.

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

PROJECTIVE TEXTURE COORDINATES

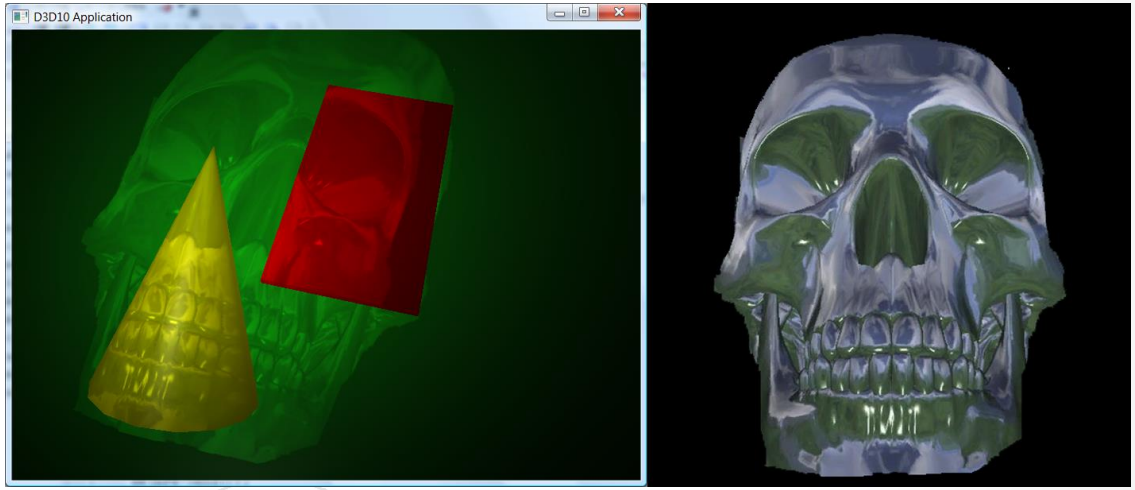
Projective texturing allows us to project a texture onto arbitrary geometry, much like a slide projector.

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

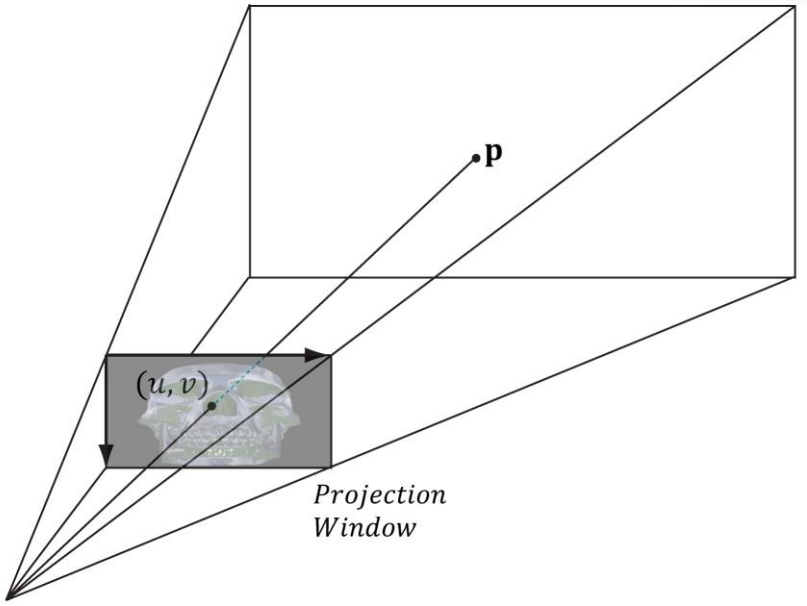
The texture coordinates (u, v) identify the texel that should be projected onto the 3D point **p**.

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 ([−1, 1] x [−1, 1]) to texture space ([0, 1] x [0, 1]) via the following change of coordinate transformation:

$$u = 0.5x + 0.5$$

$$v = -0.5y + 0.5$$

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

```
void ShadowMapApp::UpdateShadowTransform(const GameTimer& gt)
{
    // Only the first "main" light casts a shadow.
    XMVECTOR lightDir = XMLoadFloat3(&mRotatedLightDirections[0]);
    XMVECTOR lightPos = -2.0f*mSceneBounds.Radius*lightDir;
    XMVECTOR targetPos = XMLoadFloat3(&mSceneBounds.Center);
    XMVECTOR lightUp = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
    XMMATRIX lightView = XMMatrixLookAtLH(lightPos, targetPos, lightUp);

    XMStoreFloat3(&mLightPosW, lightPos);

    // Transform bounding sphere to light space.
    XMFLOAT3 sphereCenterLS;
    XMStoreFloat3(&sphereCenterLS, XMVector3TransformCoord(targetPos, lightView));

    // Ortho frustum in light space encloses scene.
    float l = sphereCenterLS.x - mSceneBounds.Radius;
    float b = sphereCenterLS.y - mSceneBounds.Radius;
    float n = sphereCenterLS.z - mSceneBounds.Radius;
    float r = sphereCenterLS.x + mSceneBounds.Radius;
    float t = sphereCenterLS.y + mSceneBounds.Radius;
    float f = sphereCenterLS.z + mSceneBounds.Radius;

    mLightNearZ = n;
    mLightFarZ = f;

    XMMATRIX lightProj = XMMatrixOrthographicOffCenterLH(l, r, b, t, n,
    f);

    // Transform NDC space [-1,+1]^2 to texture space [0,1]^2
    XMMATRIX T(
            0.5f, 0.0f, 0.0f, 0.0f,
            0.0f, -0.5f, 0.0f, 0.0f,
            0.0f, 0.0f, 1.0f, 0.0f,
            0.5f, 0.5f, 0.0f, 1.0f);

    XMMATRIX S = lightView*lightProj*T;

    XMStoreFloat4x4(&mLightView, lightView);
    XMStoreFloat4x4(&mLightProj, lightProj);

    // mShadowTransform   transforms a point P in light space to projective
    //   texture coordinate in texture space

    XMStoreFloat4x4(&mShadowTransform, S);
}
```

Code Implementation

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

// Fetch the material data.
MaterialData matData = gMaterialData[gMaterialIndex];

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

// Assumes nonuniform scaling; otherwise, need to use inverse-transpose of
world matrix.
vout.NormalW = mul(vin.NormalL, (float3x3)gWorld);

vout.TangentW = mul(vin.TangentU, (float3x3)gWorld);

// Transform to homogeneous clip space.
vout.PosH = mul(posW, gViewProj);

// Output vertex attributes for interpolation across triangle.
float4 texC = mul(float4(vin.TexC, 0.0f, 1.0f), gTexTransform);
vout.TexC = mul(texC, matData.MatTransform).xy;

// Generate projective tex-coords to project shadow map onto scene.
vout.ShadowPosH = mul(posW, gShadowTransform);

return vout;
}
```

```
float4 PS(VertexOut pin) : SV_Target
{
// Fetch the material data.
….
// 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;

// 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;
}
```

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

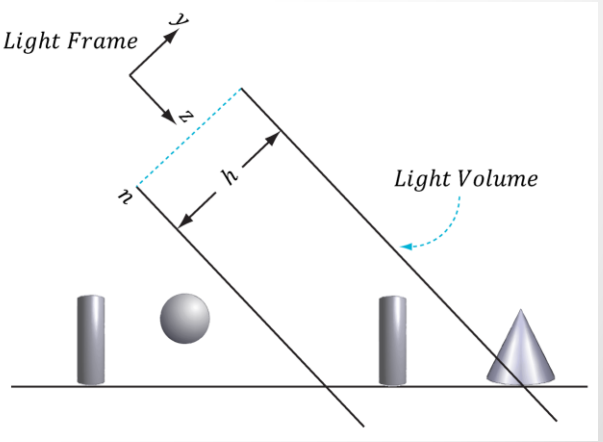
- So far we have illustrated projective texturing using perspective projections (frustum shaped volumes)
- Instead of using a perspective projection for the projection process, we could have used an orthographic projection.
- The texture is projected in the direction of the z-axis of the light through 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.
- This is faster, because it avoids the per-pixel division required for perspective projection.
- On the other hand, leaving in the division does not hurt because it divides by 1 (an orthographic projection does not change the w-coordinate, so w will be 1). If we leave the division by w in the shader code, then the shader code works for both perspective and orthographic projections uniformly.

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

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

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 th in the world.
- 6. This volume can be either a frustum volume (per box volume (orthographic projection).
- 7. A frustum light volume can be used to model sp the spotlight cone inside the frustum.
- 8. A box light volume can be used to model parall 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;

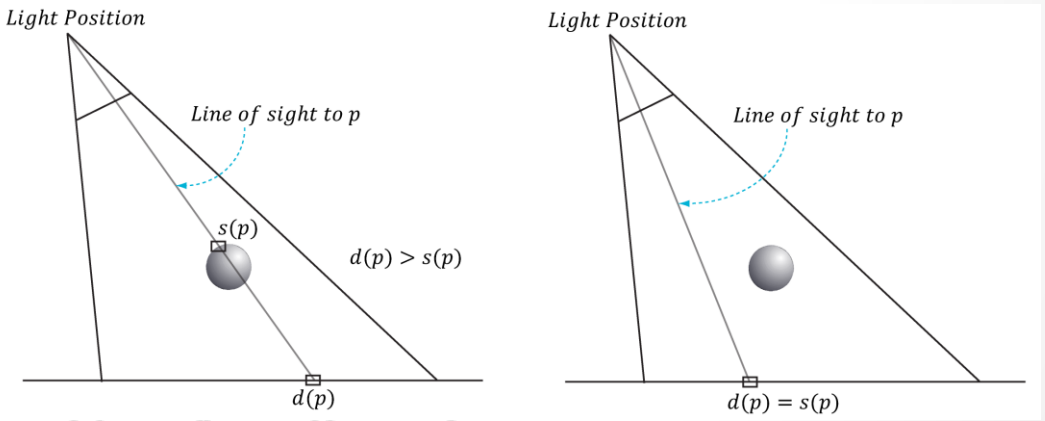
This value is the depth of the pixel closest to the light along the line of sight from the position of the light to p.

Note that a pixel p is in shadow if and only if $d(p) > s(p)$.

A pixel is not in shadow if and only if $d(p) \leq s(p)$.

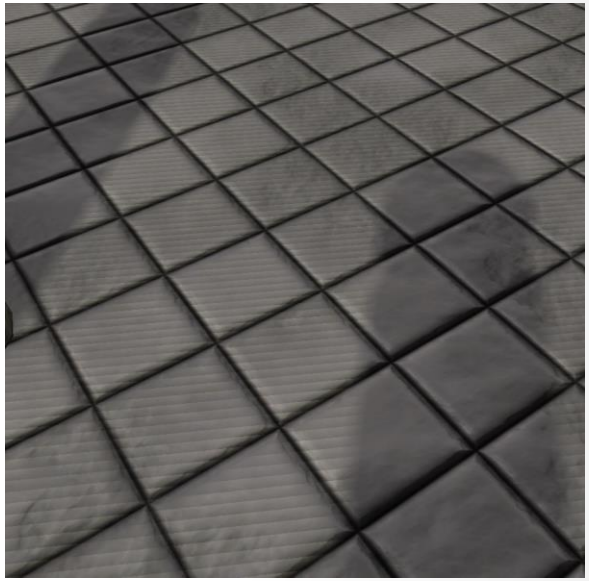
On the left, the depth of the pixel p from the light is $d(p)$. However, the depth of the pixel nearest to the light along the same line of sight has depth $s(p)$, and $d(p) > s(p)$. We conclude, therefore, that there is an object in front of p from the perspective of the light and so p is in shadow.

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.
- However, the shadow map only has some finite resolution.
- 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.
- This causes aliasing issues known as shadow acne



Shadow Acne

The eye **E** sees two points on the scene *p*1 and *p*2 that correspond to different screen pixels.

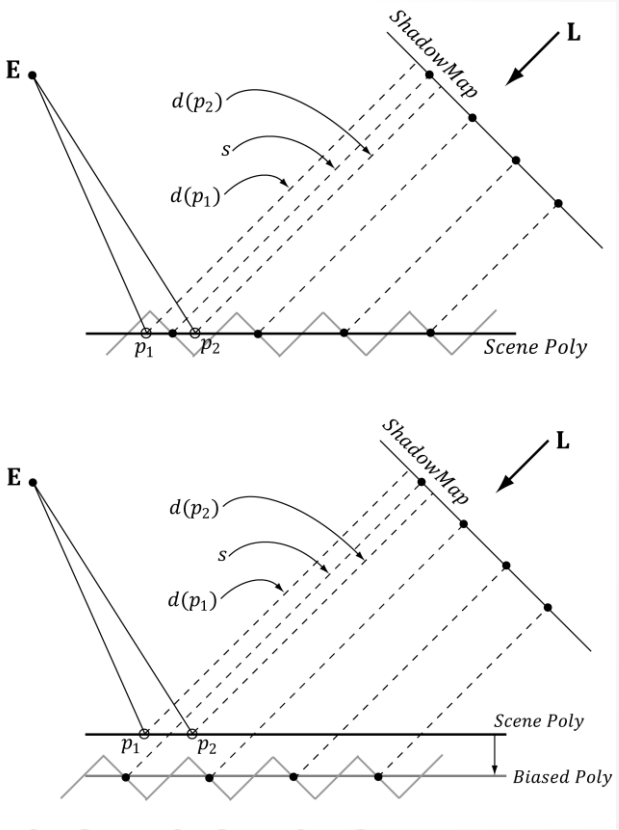
However, from the viewpoint of the light, both points are covered by the same shadow map texel (that is, *s*(*p*1) = *s*(*p*2) = *s*).

When we do the shadow map test, we have *d*(*p*1) > *s* and *d*(*p*2) ≤ *s*.

Therefore, *p*1 will be colored as if it were in shadow, and *p*2 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*(*p*1) ≤ *s* and *d*(*p*2) ≤ *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.

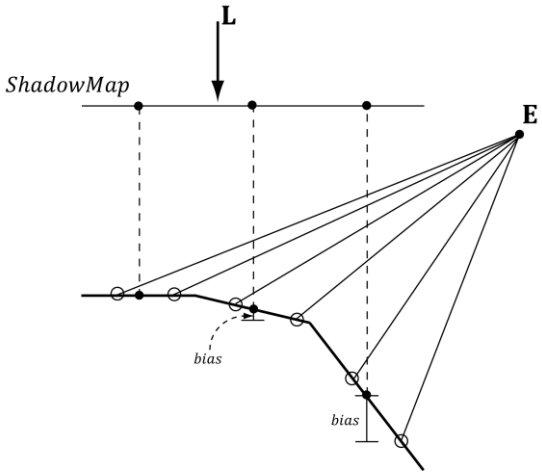
The term *Peter Panning* derives its name from a children's book character whose shadow became detached and who could fly.

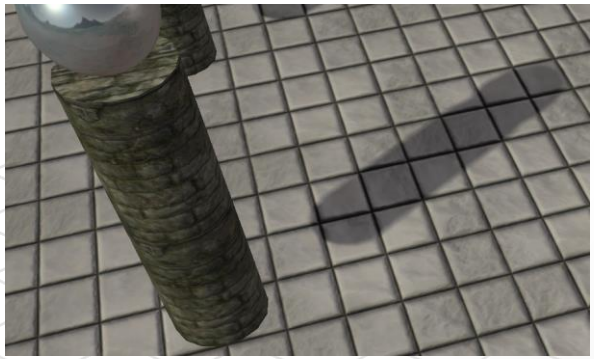
A fixed bias does not work for all geometry.

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

We need 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 output merger 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.

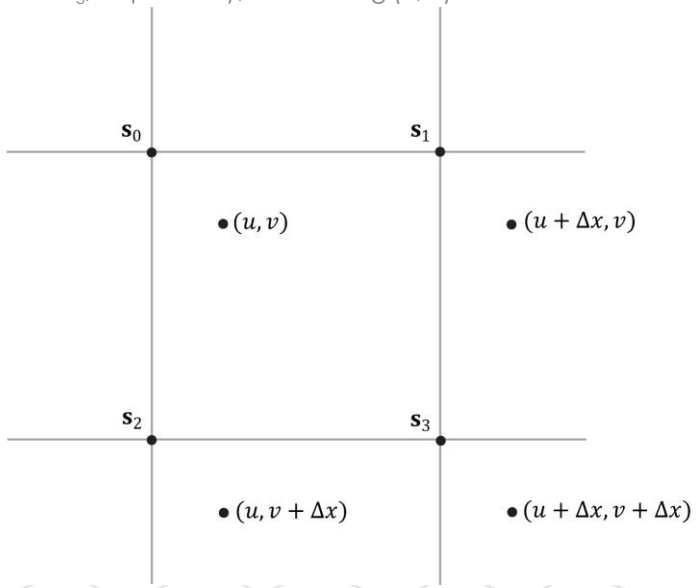
Interpolation depth values can lead to incorrect results about a pixel being flagged in shadow.

Percentage Closer Filtering (PCF) uses point filtering (MIN_MAG_MIP_POINT) and sample the texture with coordinates:

(u, v), $(u + \Delta x, v)$, $(u, v + \Delta x)$, $(u + \Delta x, v + \Delta x)$,

where Δx = 1/SHADOW_MAP_SIZE.

Since we are using point sampling, these four points will hit the nearest four texels $\mathbf{s}_0$, $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{s}_3$, respectively, surrounding (u, v).



PCF

The idea is to sample from the shadow map around the current pixel and compare its depth to all the samples. By averaging out the results we get a smoother line between light and shadow. For example, take a look at the following shadow map:

Each cell contains the depth value for each pixel (when viewed from the light source). To make life simple, let's say that the depth of all the pixels above is 0.5 (when viewed from the camera point of view). All the pixels whose shadow map value is small than 0.5 will be in shadow while the ones whose shadow map value is greater than or equal to 0.5 will be in light. This will create a hard aliased line between light and shadow.

Now consider the following - the pixels that are nearest the border between light and shadow are surrounded by pixels who shadow map value is smaller than 0.5 as well as pixels whose shadow map value is greater than or equal to 0.5.

If we sample these neighboring pixels and average out the results we will get a factor level that can help us smooth out the border between light and shadow.

Of course we don't know in advance what pixels are closest to that border so we simply do this sampling work for each pixel. This is basically the entire system.

In our demo, we sample 9 pixels in a 3 by 3 kernel around each pixel and average out the result. This will be our shadow factor instead of the 0.5 or 1.0.

0.48	0.48	0.48	0.49	0.5	0.5	0.51
0.48	0.48	0.48	0.49	0.5	0.5	0.51
0.48	0.48	0.49	0.49	0.5	0.51	0.51
0.49	0.49	0.49	0.5	0.51	0.51	0.51
0.49	0.49	0.5	0.51	0.51	0.51	0.51
0.49	0.5	0.5	0.51	0.51	0.51	0.51
0.49	0.5	0.51	0.51	0.51	0.51	0.51

The shadow map test

We 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-

```
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.

`SampleCmpLevelZero s`amples a texture and compares the result to a comparison value (*depth* in this case). This function is identical to calling **SampleCmp** on mipmap level 0 only.

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;
```

SamplerComparisonState

float Object.SampleCmpLevelZero(

SamplerComparisonState S,

float Location,

float CompareValue,

[int Offset]

);

The zero-based sampler index.

Here is a partial example of declaring sampler-comparison state, and calling a comparison sampler

```
SamplerComparisonState ShadowSampler
{
// sampler state
Filter = COMPARISON_MIN_MAG_LINEAR_MIP_POINT;
AddressU = MIRROR;
AddressV = MIRROR;

// sampler comparison state
ComparisonFunc = LESS;
};
float3 vModProjUV;
...
float fShadow = g_ShadowMap.SampleCmpLevelZero(ShadowSampler, vModProjUV.xy,
vModProjUV.z);
```

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 to build 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 and depth.

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

one of the .x, .y, .z, .w swizzle components (or the .r, .g, .b, .a equivalents) can be applied. In our case, it would be r!

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 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;

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

Note that with this setup, both $d(p)$ and $s(p)$ are measured in the NDC space of the light, so they can be compared.

The transformation matrix gShadowTransform transforms from world space to the shadow map texture space.

```
// Common.hlsl
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;
…

// Default.hlsl

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);
const float shininess = (1.0f - roughness) * normalMapSample.a;

Material mat = { diffuseAlbedo, fresnelR0, shininess };

float4 directLight = ComputeLighting(gLights, mat, pin.PosW, bumpedNormalW, toEyeW, shadowFactor);
```

LightingUtil.hlsl

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

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 in our demo. If you need to get darker shadow, you will need to add the following lines in the pixel shader.

```
shadowFactor[1] =
CalcShadowFactor(pin.ShadowPosH);

shadowFactor[2] =
CalcShadowFactor(pin.ShadowPosH);
```

```
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);
}
```

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).