Game Engine Development II

Week 8

Hooman Salamat

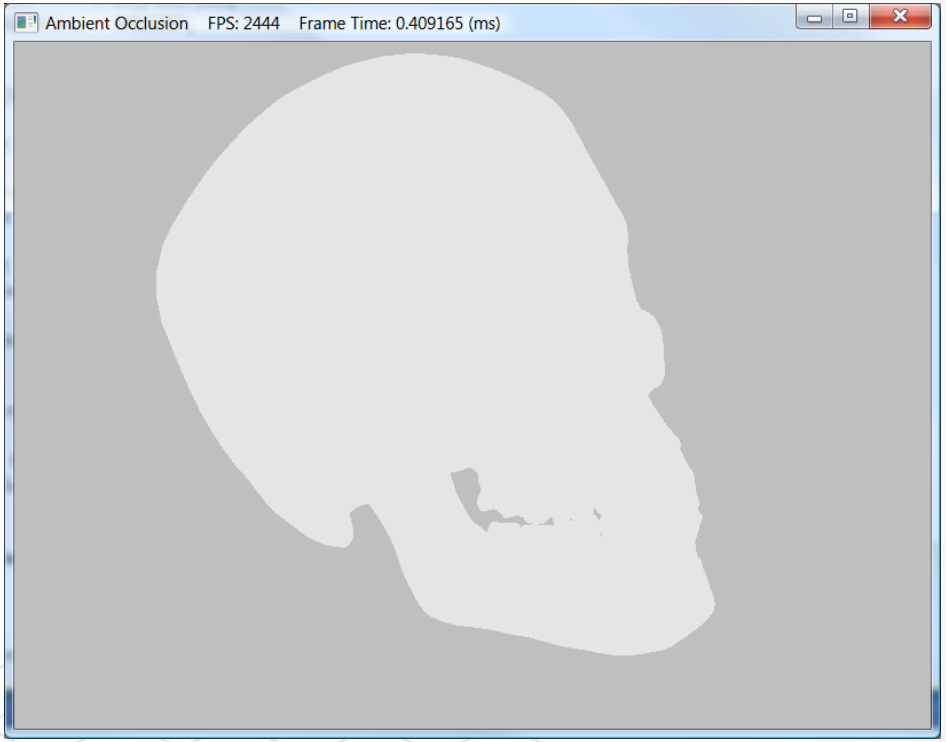
Objectives

1. To understand the basic idea behind ambient occlusion and how to implement ambient occlusion via ray casting.

2. To learn how to implement a real-time approximation of ambient occlusion in screen space called screen space ambient occlusion.

A mesh rendered with only the ambient term appears as a solid color.

We show how the screen space ambient occlusion could improve our ambient term.



Ambient Occlusion

When light travels through a scene, rebounding off surfaces, there are some places that have a smaller chance of getting hit with light: corners, tight gaps between objects, creases, etc.

This results in those areas being darker than their surroundings. This effect is called ambient occlusion (AO).

The usual method to simulate this darkening of certain areas of the scene involves testing, for each surface, how much it is "occluded" or "blocked from light" by other surfaces.

Real-time AO was out of the reach until Screen Space Ambient Occlusion (SSAO) appeared.

SSAO is a method to approximate ambient occlusion in screen space.

It was first used in games by Crytek, in their "Crysis" franchise and has been used in many other games since.

The original implementation

The original implementation by Crytek had a depth buffer as input and worked roughly like this:

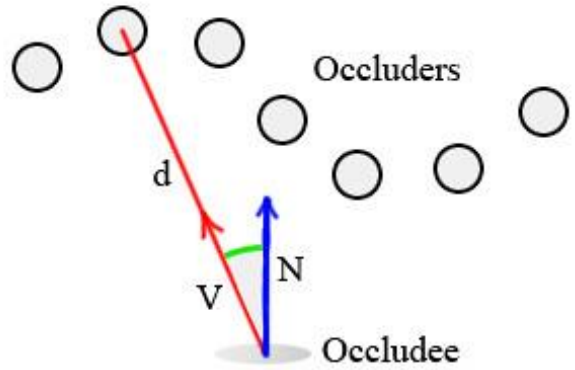
The algorithm is implemented as a pixel shader, analyzing the scene depth buffer which is stored in a texture. For every pixel on the screen, the pixel shader samples the depth values around the current pixel and tries to compute the amount of occlusion from each of the sampled points. In its simplest implementation, the occlusion factor depends only on the depth difference between sampled point and current point.

An occlusion buffer is generated by averaging the distances of occluded samples to the depth buffer.

occluders will be just points with no orientation and the occludee (the pixel which receives occlusion) will be a pair. Then, the occlusion contribution of each occluder depends on two factors:

Distance "d" to the occludee.

Angle between the occlude's normal "N" and the vector between occluder and occludee "V". With these two factors in mind, a simple formula to calculate occlusion is: Occlusion = max(0.0, dot(N, V)) * (1.0 / (1.0 + d))



AMBIENT OCCLUSION VIA RAY CASTING

Ambient light is a light type in computer graphics that is used to simulate global illumination.

Ambient occlusion is simply a simulation of the shadowing caused by objects blocking the ambient light.

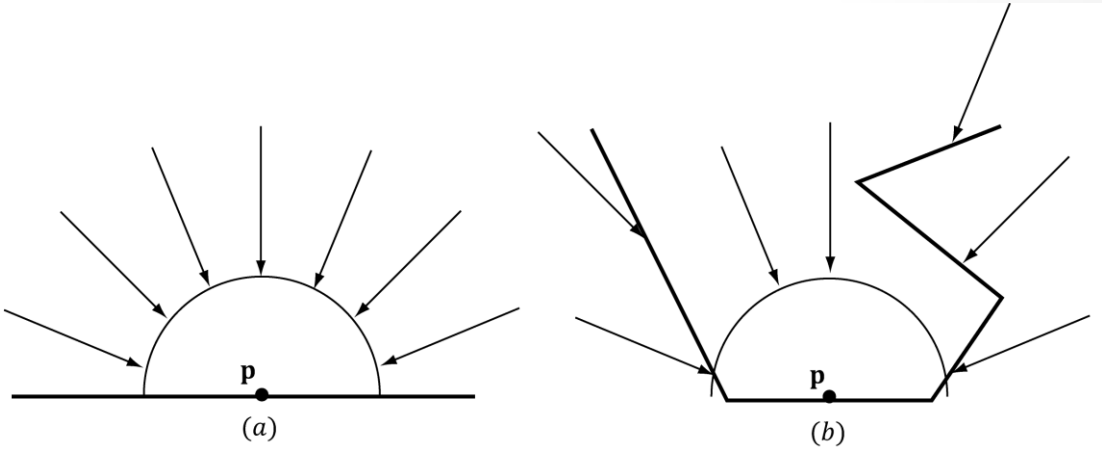
Because ambient light is environmental, unlike other types of lighting, ambient occlusion does not depend on light direction. As such, it can be pre-computed for static objects.

The amount of indirect light a point **p** on a surface receives is proportional to how occluded it is to incoming light over the hemisphere about **p**.

(a) A point **p** is completely unoccluded and all incoming light over the hemisphere about **p** reaches **p**.

(b) Geometry partially occludes **p** and blocks incoming light rays over the hemisphere about **p**.

One way to estimate the occlusion of a point **p** is via ray casting. We randomly cast rays over the hemisphere about **p**, and check for intersections against the mesh.



RAY CASTING

If we cast *N* rays, and *h* of them intersect the mesh, then the point has the occlusion value:

$$\boldsymbol{occlusion} = \frac{\boldsymbol{h}}{\boldsymbol{N}} \in [\mathbf{0}, \mathbf{1}]$$

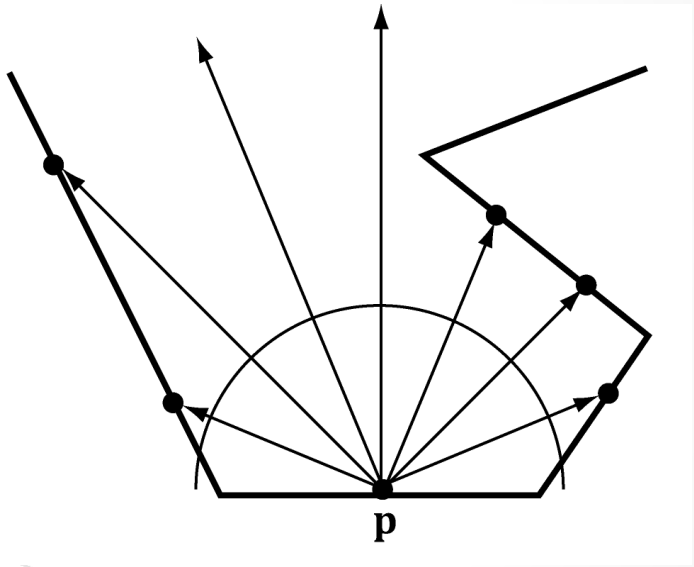
Only rays with an intersection point **q** whose distance from **p** is less than some threshold value *d* should contribute to the occlusion estimate;

An intersection point **q** far away from **p** is too far to occlude it.

The occlusion factor measures how occluded the point is (i.e., how much light it does not receive).

The "accessibility" (ambient-access) determines how much light a point does receive and is derived from occlusion as:

$$\boldsymbol{accessibility} = \mathbf{1} - \boldsymbol{occlusion} \in [\mathbf{0}, \mathbf{1}]$$



Octree

An octree is a tree data structure in which each internal node has exactly eight children (octants).

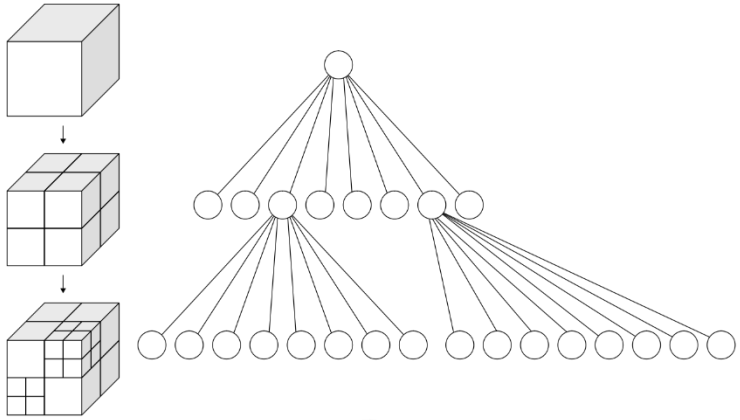
The node stores an explicit three-dimensional point, which is the "center" of the subdivision for that node;

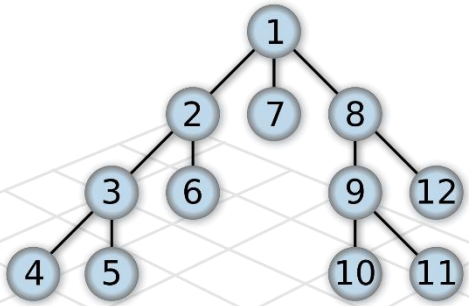
Octrees are most often used to partition a three-dimensional space by recursively subdividing it into eight octants.

Octrees are the three-dimensional analog of quadtrees.

Octrees are often used in 3D graphics and 3D game engines.

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.





BuildVertexAmbientOcclusion

```
void AmbientOcclusionApp::BuildVertexAmbientOcclusion(
std::vector<Vertex::AmbientOcclusion>& vertices,
const std::vector<UINT>& indices)
{
UINT vcount = vertices.size();
UINT tcount = indices.size() / 3;
std::vector<XMFLOAT3> positions(vcount);
for (UINT i = 0; i < vcount; ++i)
positions[i] = vertices[i].Pos;
Octree octree;
octree.Build(positions, indices);
// For each vertex, count how many triangles contain the vertex.
std::vector<int> vertexSharedCount(vcount);
// Cast rays for each triangle, and average triangle occlusion
// with the vertices that share this triangle.
for (UINT i = 0; i < tcount; ++i)
{
UINT i0 = indices[i * 3 + 0];
UINT i1 = indices[i * 3 + 1];
UINT i2 = indices[i * 3 + 2];
XMVECTOR v0 = XMLoadFloat3(&vertices[i0].Pos);
XMVECTOR v1 = XMLoadFloat3(&vertices[i1].Pos);
XMVECTOR v2 = XMLoadFloat3(&vertices[i2].Pos);
XMVECTOR edge0 = v1 - v0;
XMVECTOR edge1 = v2 - v0;
XMVECTOR normal = XMVector3Normalize(
XMVector3Cross(edge0, edge1));
XMVECTOR centroid = (v0 + v1 + v2) / 3.0f;
```

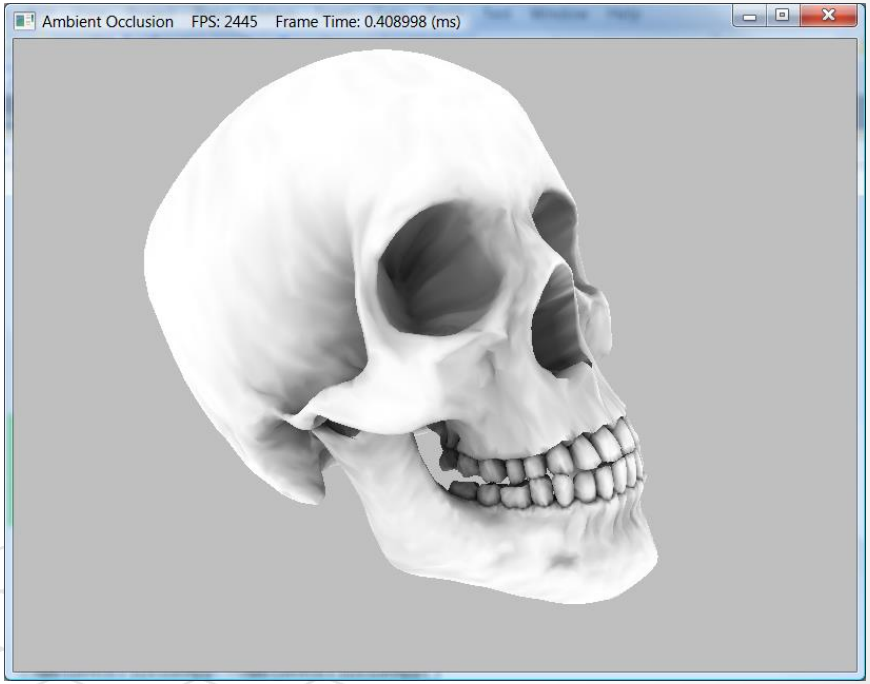
```
// Offset to avoid self intersection.
centroid += 0.001f * normal;
const int NumSampleRays = 32;
float numUnoccluded = 0;

for (int j = 0; j < NumSampleRays; ++j)
{
XMVECTOR randomDir =
MathHelper::RandHemisphereUnitVec3(normal);

// Test if the random ray intersects the scene mesh.
if (!octree.RayOctreeIntersect(centroid, randomDir))
{
numUnoccluded++;
}
}
float ambientAccess = numUnoccluded /NumSampleRays;
// Average with vertices that share this face.
vertices[i0].AmbientAccess += ambientAccess;
vertices[i1].AmbientAccess += ambientAccess;
vertices[i2].AmbientAccess += ambientAccess;
vertexSharedCount[i0]++;
vertexSharedCount[i1]++;
vertexSharedCount[i2]++;
}
// Finish average by dividing by the number of samples we added,
// and store in the vertex attribute.
for (UINT i = 0; i < vcount; ++i)
{
vertices[i].AmbientAccess /= vertexSharedCount[i];
}
}
```

Implementation

The code performs the ray cast per triangle, and then averages the occlusion results with the vertices that share the triangle. The ray origin is the triangle's centroid, and we generate a random ray direction over the hemisphere of the triangle.

The mesh is rendered only with ambient occlusion—there are no scene lights. Notice how the crevices are darker; this is because when we cast rays out they are more likely to intersect geometry and contribute to occlusion. On the other hand, the skull cap is white (unoccluded) because when we cast rays out over the hemisphere for points on the skull cap, they will not intersect any geometry of the skull.

The code uses an octree to speed up the ray/triangle intersection tests. For a mesh with thousands of triangles, it would be very slow to test each random ray with every mesh triangle. An octree sorts the triangles spatially, so we can quickly find only the triangles that have a good chance of intersecting the ray; this reduces the number of ray/triangle intersection tests substantially.



xNormal

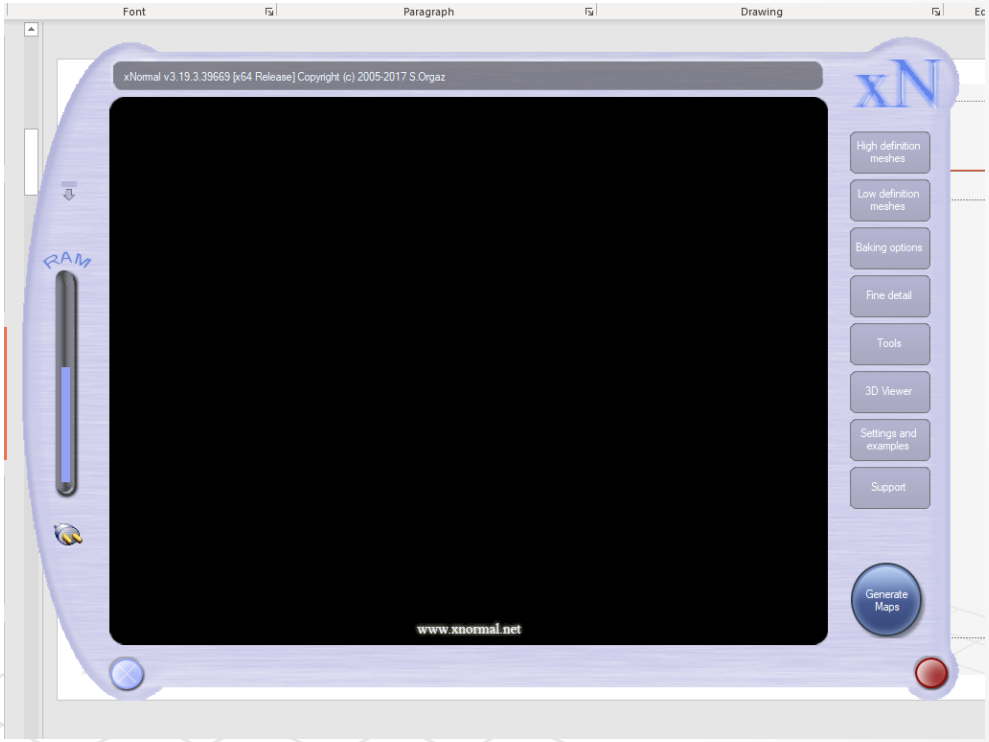
Precomputing ambient occlusion works well for static models.

xNormal™ is a free app to bake texture maps (like *normal maps* and *ambient occlusion*).

http://www.xnormal.net *xNormal* generates *ambient occlusion maps*—textures that store ambient occlusion data.

However, casting rays at runtime to implement dynamic ambient occlusion is not feasible.

In the next section, we examine a popular technique for computing ambient occlusion in real-time using screen space information.



Screen Space Ambient Occlusion (SSAO)

The strategy of *screen space ambient occlusion* (SSAO) is, for every frame, render the scene view space normals to a full screen render target and the scene depth to the usual depth/stencil buffer, and then estimate the ambient occlusion at each pixel using only the view space normal render target and the depth/stencil buffer as input.

Once we have a texture that represents the ambient occlusion at each pixel, we render the scene as usual to the back buffer, but apply the SSAO information to scale the ambient term at each pixel.

The strategy of *screen space ambient occlusion*

- 1. Render Normals and Depth Pass
- 2. Ambient Occlusion Pass
- 2.1 Reconstruct View Space Position
- 2.2 Generate Random Samples
- 2.3 Generate the potential occluding points
- 2.4 Perform the Occlusion test
- 2.5 Apply the SSAO information to scale the ambient term at each pixel

Render Normals and Depth Pass

First, we render the view space normal vectors of the scene objects to a screen sized DXGI_FORMAT_R16G16B16A16_FLOAT texture map, while the usual depth/stencil buffer is bound to lay down the scene depth.

The vertex/pixel shaders used for this pass are in DrawNormals.hlsl.

The pixel shader outputs the normal vector in view space.

Notice that we are writing to a floating-point render target, so there is no problem writing out arbitrary floating-point data.

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

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

// 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.
float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);
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;
}

float4 PS(VertexOut pin) : SV_Target
{
// Fetch the material data.
MaterialData matData = gMaterialData[gMaterialIndex];
float4 diffuseAlbedo = matData.DiffuseAlbedo;
uint diffuseMapIndex = matData.DiffuseMapIndex;
uint normalMapIndex = matData.NormalMapIndex;
    // Dynamically look up the texture in the array
diffuseAlbedo *= gTextureMaps[diffuseMapIndex].Sample(gsamAnisotropicWrap, pin.TexC);
#ifdef ALPHA_TEST
clip(diffuseAlbedo.a - 0.1f);
#endif
// Interpolating normal can unnormalize it, so renormalize it.
    pin.NormalW = normalize(pin.NormalW);

    // NOTE: We use interpolated vertex normal for SSAO. Write normal in view space coordinates
    float3 normalV = mul(pin.NormalW, (float3x3)gView);
    return float4(normalV, 0.0f);
}
```

ComputeSsao

```
void ComputeSsao(

ID3D12GraphicsCommandList* cmdList,

FrameResource* currFrame,

int blurCount);
```

Changes the render target to the Ambient render target and draws a full screen quad to kick off the pixel shader to compute the AmbientMap.

We still keep the main depth buffer binded to the pipeline, but depth buffer read/writes are disabled, as we do not need the depth buffer computing the Ambient map.

```
mCommandList->SetGraphicsRootSignature(mSsaoRootSignature.Get());

mSsao->ComputeSsao(mCommandList.Get(), mCurrFrameResource, 3);
```

Ambient Occlusion Pass

- After we have laid down the view space normals and scene depth
- 1. Disable the depth buffer : we do not need it for generating the ambient occlusion texture
- 2. Draw a full screen quad to invoke the SSAO pixel shader at each pixel.
- 3. The pixel shader will then use the normal texture and depth buffer to generate an ambient accessibility value at each pixel.
- We call the generated texture map in this pass the *SSAO map*.
- 4. We render the normal/depth map at full screen resolution to the SSAO map at half the width and height of the back buffer for performance reasons.

SSAO - Ambient Occlusion pass

The points involved in SSAO:

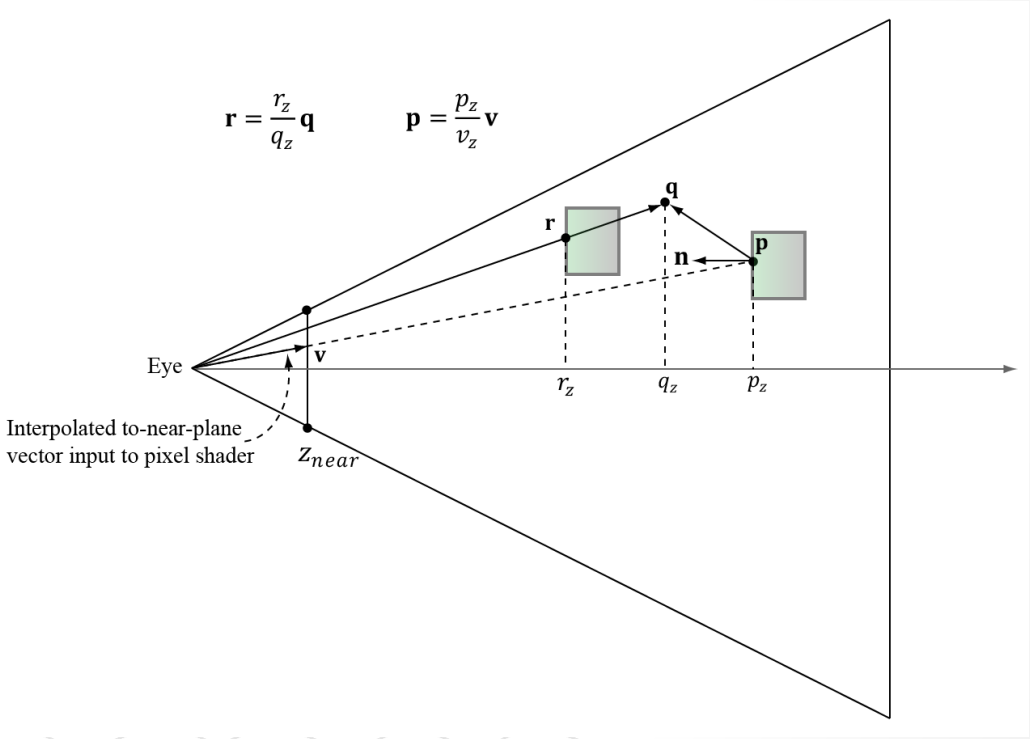
The point **p** corresponds to the current pixel we are processing, and it is reconstructed from the depth value stored in the depth buffer and the vector **v** that passes through the near plane at this pixel.

The point **q** is a random point in the hemisphere of **p**.

The point **r** corresponds to the nearest visible point along the ray from the eye to **q**.

The point **r** contributes to the occlusion of **p** if $|p_z - r_z|$ is sufficiently small and the angle between **r** − **p** and **n** is less than 90°.

In the demo, we take 14 random sample points and average the occlusion from each to estimate the ambient occlusion in screen space.



Reconstruct View Space Position

When we draw the full screen quad to invoke the SSAO pixel shader at each pixel of the SSAO map, we can use the inverse of the projection matrix to transform the quad corner points in NDC space to points on the near projection plane window:

```
// Ssao.hlsl

static const float2 gTexCoords[6] =
{
    float2(0.0f, 1.0f),
    float2(0.0f, 0.0f),
    float2(1.0f, 0.0f),
    float2(0.0f, 1.0f),
    float2(1.0f, 0.0f),
    float2(1.0f, 1.0f)
};
```

```
struct VertexOut
{
float4 PosH : SV_POSITION;
float3 PosV : POSITION;
float2 TexC : TEXCOORD0;
};

VertexOut VS(uint vid : SV_VertexID)
{
    VertexOut vout;

    vout.TexC = gTexCoords[vid];

    // Quad covering screen in NDC space.
    vout.PosH = float4(2.0f*vout.TexC.x - 1.0f, 1.0f - 2.0f*vout.TexC.y, 0.0f, 1.0f);

    // Transform quad corners to view space near plane.
    float4 ph = mul(vout.PosH, gInvProj);
    vout.PosV = ph.xyz / ph.w;

    return vout;
}
```

Reconstruct View Space Position

These to-near-plane vectors are interpolated across the quad and give us a vector **v** from the eye to the near plane for each pixel.

For each pixel, we sample the depth buffer so that we have the z-coordinate p_z of the nearest visible point to the eye in NDC coordinates.

The goal is to reconstruct the view space position **p** = (p_x, p_y, p_z) from the sampled NDC z-coordinate p_z and the interpolated to-near-plane vector **v**.

Since the ray of **v** passes through **p**, there exists a *t* such that **p** = *t***v**.

In particular, $p_z = tv_z$ so that $t = \frac{p_z}{v_z}$.

Therefore: $\boldsymbol{p} = \frac{p_z}{v_z}\,\boldsymbol{v}$

The reconstruction code in the pixel shader is as follows:

```
// Ssao.hlsl
float NdcDepthToViewDepth(float z_ndc)
{
// We can invert the calculation from NDC space to view space for the  z-coordinate.
// We have that  z_ndc = A + B/viewZ, where gProj[2,2]=A and gProj[3, 2] = B.
// Therefore: z_ndc = A + B/viewZ, where gProj[2,2]=A and gProj[3,2]=B.
    float viewZ = gProj[3][2] / (z_ndc - gProj[2][2]);
    return viewZ;
}

float4 PS(VertexOut pin) : SV_Target
{
// p -- the point we are computing the ambient occlusion for.
// n -- normal vector at p.
// q -- a random offset from p.
// r -- a potential occluder that might occlude p.

// Get viewspace normal and z-coord of this pixel.
    float3 n = normalize(gNormalMap.SampleLevel(gsamPointClamp, pin.TexC, 0.0f).xyz);
    float pz = gDepthMap.SampleLevel(gsamDepthMap, pin.TexC, 0.0f).r;
    pz = NdcDepthToViewDepth(pz);
```

Generate Random Samples

This step is analogous to the random ray cast over the hemisphere.

We randomly sample *N* points **q** about **p** that are also in front of **p** and within a specified occlusion radius.

The occlusion controls how far away from **p** we want to take the random sample points.

Choosing to only sample points in front of **p** is analogous to only casting rays over the hemisphere instead of the whole sphere when doing ray casted ambient occlusion.

We can generate random vectors and store them in a texture map, and then sample this texture map at *N* different positions to get *N* random vectors.

In our implementation, we generate fourteen equally distributed vectors (since they are random, this prevents samples to clump together in roughly the same direction)

```
void Ssao::BuildOffsetVectors()
{
// Start with 14 uniformly distributed vectors.  We choose the 8 corners of the cube
// and the 6 center points along each cube face.  We always alternate the points on
// opposites sides of the cubes.  This way we still get the vectors spread out even
// if we choose to use less than 14 samples.
mOffsets[0] = XMFLOAT4(+1.0f, +1.0f, +1.0f, 0.0f);
mOffsets[1] = XMFLOAT4(-1.0f, -1.0f, -1.0f, 0.0f);

mOffsets[2] = XMFLOAT4(-1.0f, +1.0f, +1.0f, 0.0f);
mOffsets[3] = XMFLOAT4(+1.0f, -1.0f, -1.0f, 0.0f);

mOffsets[4] = XMFLOAT4(+1.0f, +1.0f, -1.0f, 0.0f);
mOffsets[5] = XMFLOAT4(-1.0f, -1.0f, +1.0f, 0.0f);

mOffsets[6] = XMFLOAT4(-1.0f, +1.0f, -1.0f, 0.0f);
mOffsets[7] = XMFLOAT4(+1.0f, -1.0f, +1.0f, 0.0f);

// 6 centers of cube faces
mOffsets[8] = XMFLOAT4(-1.0f, 0.0f, 0.0f, 0.0f);
mOffsets[9] = XMFLOAT4(+1.0f, 0.0f, 0.0f, 0.0f);

mOffsets[10] = XMFLOAT4(0.0f, -1.0f, 0.0f, 0.0f);
mOffsets[11] = XMFLOAT4(0.0f, +1.0f, 0.0f, 0.0f);

mOffsets[12] = XMFLOAT4(0.0f, 0.0f, -1.0f, 0.0f);
mOffsets[13] = XMFLOAT4(0.0f, 0.0f, +1.0f, 0.0f);

    for(int i = 0; i < 14; ++i)
{
// Create random lengths in [0.25, 1.0].
float s = MathHelper::RandF(0.25f, 1.0f);

XMVECTOR v = s * XMVector4Normalize(XMLoadFloat4(&mOffsets[i]));

XMStoreFloat4(&mOffsets[i], v);
}
}
```

Generate the Potential Occluding Points

- We now have random sample points **q** surrounding **p**. However, we know nothing about them—whether they occupy empty space or a solid object; therefore, we cannot use them to test if they occlude **p**.
- To find potential occluding points, we need depth information from the depth buffer.
- We generate projective texture coordinates for each **q** with respect to the camera, and use these to sample the depth buffer to get the depth in NDC space.
- Then we transform to view space to obtain the depth r_z of the nearest visible pixel along the ray from the eye to **q**.
- With the z-coordinates r_z known, we can reconstruct the full 3D view space position **r.**
- Because the vector from the eye to **q** passes through **r** there exists a *t* such that $\boldsymbol{r} = t\boldsymbol{q}$. In particular, $r_z = tq_z$ so $t = \frac{r_z}{q_z}$.
- Therefore $\boldsymbol{r} = \frac{r_z}{q_z}\,\boldsymbol{q}$
- The points **r**, one generated for each random sample point **q**, are our potential occluding points.

Perform the Occlusion Test

Now that we have our potential occluding points **r**, we can perform our occlusion test to estimate if they occlude **p**. The test relies on two quantities:

1. The view space depth distance $|p_z - r_z|$. We linearly scale down the occlusion as the distance increases since points farther away from have less of an occluding effect. If the distance is beyond some specified maximum distance, then no occlusion occurs.

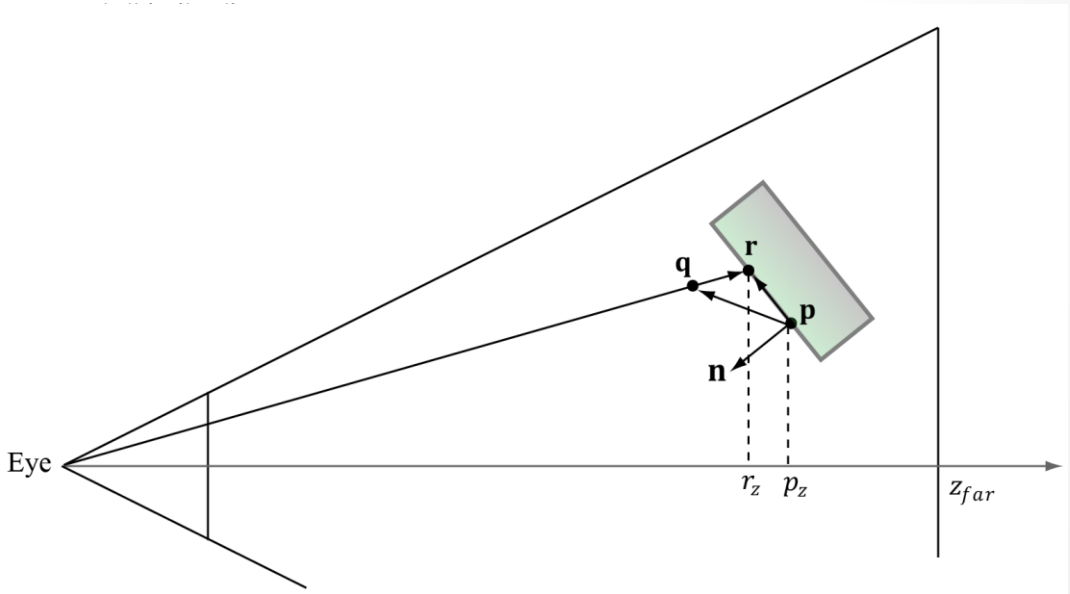
Also, if the distance is very small, then we assume **p** and **q** are on the same plane so **q** cannot occlude **p**.

2. The angle between **n** and **r** − **p** measured by

$$\max\left(\boldsymbol{n}.\left(r - \frac{p}{||r-p||}\right), 0\right)$$

This is to prevent self-intersection

If **r** lies on the same plane as **p**, it can pass the first condition that the distance $|p_z - r_z|$ is small enough that **r** occludes **p**. However, the figure shows this is incorrect as **r** does not occlude **p** since they lie on the same plane. Scaling the $\max\left(\boldsymbol{n}.\left(r - \frac{p}{||r-p||}\right), 0\right)$ occlusion by



Finishing the Calculation

After we have summed the occlusion from each sample

1. we compute the average occlusion by dividing by the sample count.

2. Then we compute the ambient-access,

3. Finally raise the ambient-access to a power to increase the contrast.

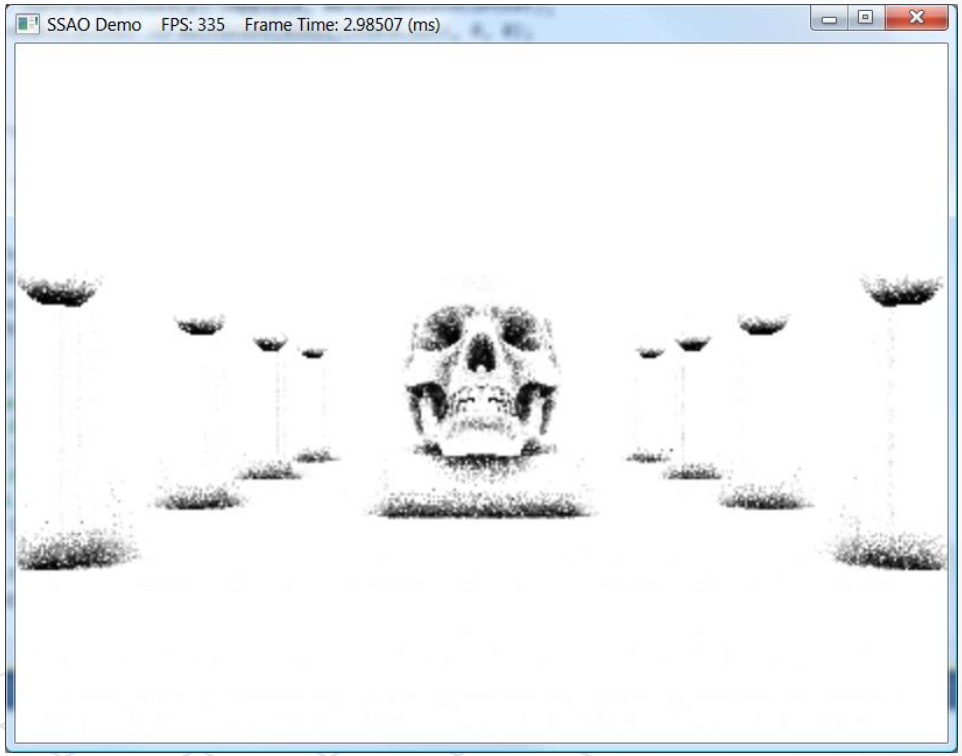
You may also wish to increase the brightness of the ambient map by adding some number to increase the intensity. You can experiment with different contrast/brightness values.

```
// Ssao.hlsl

float4 PS(VertexOut pin) : SV_Target

{

……

occlusionSum /= gSampleCount;


float access = 1.0f - occlusionSum;


// Sharpen the contrast of the SSAO map to make the SSAO affect more dramatic.

return saturate(pow(access, 6.0f));

}
```

Implementation

SSAO appears noisy due to the fact that we have only taken a few random samples.

For scenes with large viewing distances, rendering errors can result due to the limited accuracy of the depth buffer. A simple solution is to fade out the affect of SSAO with distance.



Blur Pass

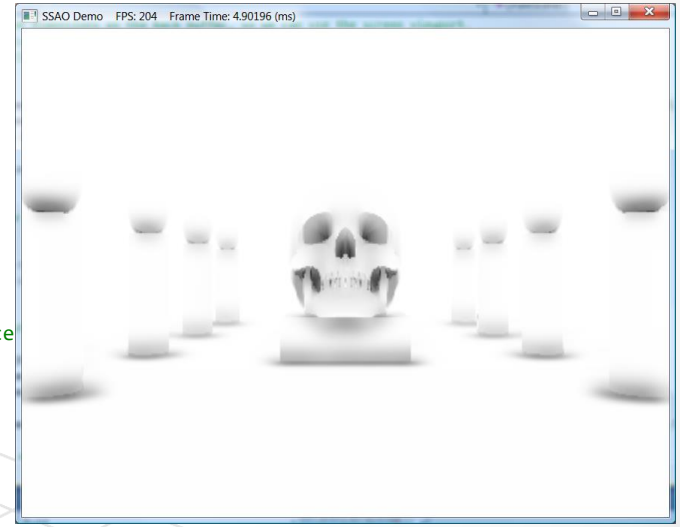
Taking enough samples to hide the noise is impractical for real-time.

The common solution is to apply an edge preserving blur (i.e., bilateral blur) to the SSAO map to smooth it out.

If we used a nonedged preserving blur, then we lose definition in the scene as sharp discontinuities become smoothed out.

The edge preserving blur is similar to the blur we implemented before, except we add a conditional statement so that we do not blur across edges (edges are detected from the normal/depth map):

```
//=============================================================================
// SsaoBlur.hlsl
// Performs a bilateral edge preserving blur of the ambient map.  We use
// a pixel shader instead of compute shader to avoid the switch from
// compute mode to rendering mode.  The texture cache makes up for some of the
// loss of not having shared memory.  The ambient map uses 16-bit texture
// format, which is small, so we should be able to fit a lot of texels
// in the cache.
//=============================================================================

cbuffer cbSsao : register(b0)
{
    float4x4 gProj;
    float4x4 gInvProj;
    float4x4 gProjTex;
    float4   gOffsetVectors[14];

    // For SsaoBlur.hlsl
    float4 gBlurWeights[3];

    float2 gInvRenderTargetSize;

    // Coordinates given in view space
    float gOcclusionRadius;
    float gOcclusionFadeStart;
    float gOcclusionFadeEnd;
    float gSurfaceEpsilon;


};

cbuffer cbRootConstants : register(b1)
{
    bool gHorizontalBlur;
```



Blur Pass Shader

```
float4 PS(VertexOut pin) : SV_Target
{
    // unpack into float array.
    float blurWeights[12] =
    {
        gBlurWeights[0].x, gBlurWeights[0].y, gBlurWeights[0].z, gBlurWeights[0].w,
        gBlurWeights[1].x, gBlurWeights[1].y, gBlurWeights[1].z, gBlurWeights[1].w,
        gBlurWeights[2].x, gBlurWeights[2].y, gBlurWeights[2].z, gBlurWeights[2].w,
    };

float2 texOffset;
if(gHorizontalBlur)
{
texOffset = float2(gInvRenderTargetSize.x, 0.0f);
}
else
{
texOffset = float2(0.0f, gInvRenderTargetSize.y);
}

// The center value always contributes to the sum.
float4 color      = blurWeights[gBlurRadius] * gInputMap.SampleLevel(gsamPointClamp, pin.TexC, 0.0);
float totalWeight = blurWeights[gBlurRadius];

float3 centerNormal = gNormalMap.SampleLevel(gsamPointClamp, pin.TexC, 0.0f).xyz;
float  centerDepth = NdcDepthToViewDepth(
        gDepthMap.SampleLevel(gsamDepthMap, pin.TexC, 0.0f).r);

for(float i = -gBlurRadius; i <=gBlurRadius; ++i)
{
// We already added in the center weight.
if( i == 0 )
continue;

float2 tex = pin.TexC + i*texOffset;

float3 neighborNormal = gNormalMap.SampleLevel(gsamPointClamp, tex, 0.0f).xyz;
float  neighborDepth  = NdcDepthToViewDepth(gDepthMap.SampleLevel(gsamDepthMap, tex, 0.0f).r);

// If the center value and neighbor values differ too much (either in
// normal or depth), then we assume we are sampling across a discontinuity.
// We discard such samples from the blur.

if( dot(neighborNormal, centerNormal) >= 0.8f &&
    abs(neighborDepth - centerDepth) <= 0.2f )
{
            float weight = blurWeights[i + gBlurRadius];

//weight = 0.0f; // commenting out this line removes the blur effect

// Add neighbor pixel to blur.
color += weight*gInputMap.SampleLevel(
                gsamPointClamp, tex, 0.0);

totalWeight += weight;
}
}

// Compensate for discarded samples by making total weights sum to 1.
    return color / totalWeight;

}
```

Using the Ambient Occlusion Map

The final step is to apply the ambient occlusion map to the scene.

You might think to use alpha blending and modulate the ambient map with the back buffer. However, if we do this, then the ambient map modifies not just the ambient term, but also the diffuse and specular term of the lighting equation, which is incorrect. Instead,

1) When we render the scene to the back buffer, we bind the ambient map as a shader input.

2) We then generate projective texture coordinates (with respect to the camera),

3) Sample the SSAO map,

4) Apply the sample only to the ambient term of the lighting equation.

```
VertexOut VS(VertexIn vin)
{
…..
    // Transform to homogeneous clip space.
    vout.PosH = mul(posW, gViewProj);

    // Generate projective tex-coords to project SSAO map onto scene.
 vout.SsaoPosH = mul(posW, gViewProjTex);


float4 PS(VertexOut pin) : SV_Target
{
// Fetch the material data.
……
// Uncomment to turn off normal mapping.
    //bumpedNormalW = pin.NormalW;

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

    // Finish texture projection and sample SSAO map.
    pin.SsaoPosH /= pin.SsaoPosH.w;
    float ambientAccess = gSsaoMap.Sample(gsamLinearClamp, pin.SsaoPosH.xy,
0.0f).r;

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

Demo

Figure shows the scene with the SSAO map applied.

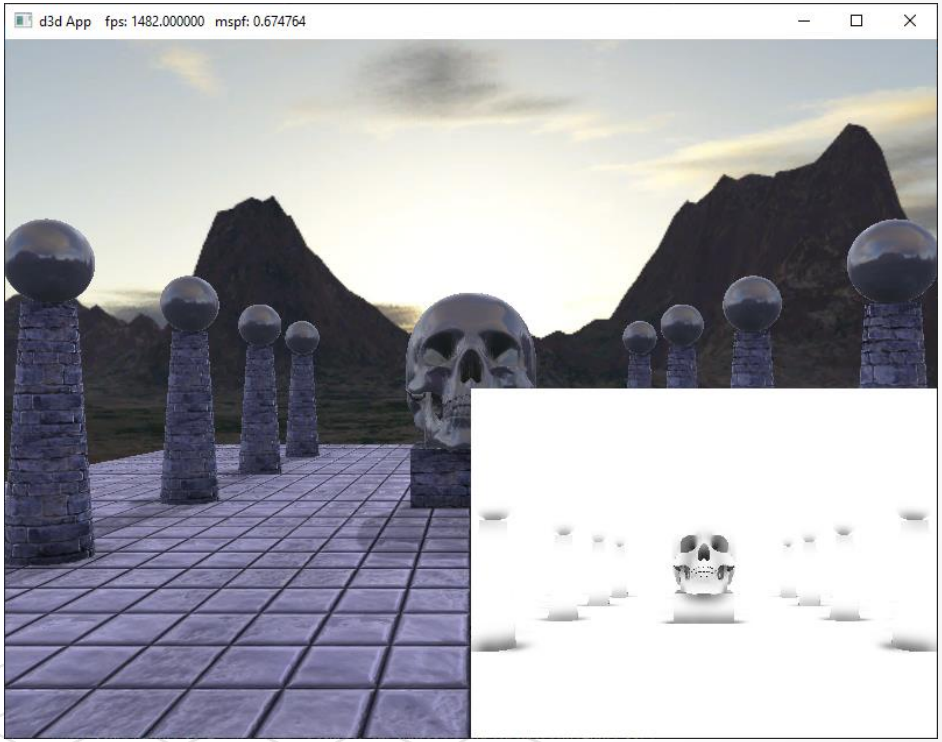
The SSAO can be subtle, and your scene has to reflect enough ambient light so that scaling it by the ambient-access makes enough of a noticeable difference.

The advantage of SSAO is most apparent when objects are in shadow.

For when objects are in shadow, the diffuse and specular terms are killed; thus only the ambient term shows up.

Without SSAO, objects in shadow will appear flatly lit by a constant ambient term, but with SSAO they will keep their 3D definition.

The affects are subtle as they only affect the ambient term, but you can see darkening at the base of the columns and box, under the spheres, and around the skull.



The depth buffer for the scene

When we render the scene view space normals, we also build the depth buffer for the scene.

Consequently, when we render the scene the second time with the SSAO map, we modify the depth comparison test to “EQUALS.”

This prevents any overdraw in the second rendering pass, as only the nearest visible pixels will pass this depth comparison test.

Moreover, the second rendering pass does not need to write to the depth buffer because we already wrote the scene to the depth buffer in the normal render target pass.

```
// PSO for opaque objects.


    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc = basePsoDesc;

    opaquePsoDesc.DepthStencilState.DepthFunc = D3D12_COMPARISON_FUNC_EQUAL;

    opaquePsoDesc.DepthStencilState.DepthWriteMask = D3D12_DEPTH_WRITE_MASK_ZERO;

    ThrowIfFailed(md3dDevice->CreateGraphicsPipelineState(&opaquePsoDesc,
IID_PPV_ARGS(&mPSOs["opaque"])));
```

Further Readings

- https://www.gamedev.net/articles/programming/graphics/a-simple-and-practical-approach-to-ssao-r2753/
- http://www.iquilezles.org/www/articles/ssao/ssao.htm
- http://www.richardssoftware.net/2013/11/ssao-with-slimdx-and-directx11.html
- http://john-chapman-graphics.blogspot.ca/2013/01/ssao-tutorial.html?m=1

Menus

Objectives

- Implement screens and menus
- Design a UI component hierarchy
- Implement containers, labels and buttons

GUI

- In most cases, the mouse will be used as an input source
  - Triggers button clicks or mouse-overs

- In the ongoing example, the keyboard is used to change menu options
- A GUI framework will be created and the first step is to reserve/create a namespace called `GUI`
- In this namespace we are defining a class called `Component` as shown on the next slide

Dangling pointer

- Problem with dangling pointer
- ref will point to undefined data!

```
int* ptr = new int(10);
int* ref = ptr;
delete ptr;
```

shared_ptr & weak_ptr

- std::shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object.
- std::weak_ptr is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by std::shared_ptr.

```
// empty definition
std::shared_ptr<int> sptr;

// takes ownership of pointer
sptr.reset(new int);
*sptr = 10;

// get pointer to data without taking ownership
std::weak_ptr<int> weak1 = sptr;
```

lock()

- shared_ptr : holds the real object.
- weak_ptr : uses lock to connect to the real owner or returns a NULL shared_ptr otherwise.
- weak_ptr role is similar to the role of real estate agent.
- Without agents, to buy a house, we may have to check random houses in the city.
- The agents make sure that we visit only those houses which are still accessible and available to buy.

```
// deletes managed object, acquires new pointer
    std::shared_ptr<int> sptr;
    sptr.reset(new int);
    *sptr = 5;

    // get pointer to new data without taking
ownership

    std::shared_ptr<int> weak1 = sptr;

    std::weak_ptr<int> weak2 = sptr;

    // weak1 is expired!
    if (auto tmp = weak1.lock())
        std::cout << *tmp << '\n';
    else
        std::cout << "weak1 is expired\n";

    // weak2 points to new data (5)
    if (auto tmp = weak2.lock())
        std::cout << *tmp << '\n';
    else
        std::cout << "weak2 is expired\n";
```

shared_ptr use case

- Suppose you have Game and Scene objects.
- The Game object will have pointers to its Scene objects.
- And it's likely that the Scene objects will also have a back pointer to their Game object.
- Then you have a dependency cycle. If you use shared_ptr, objects will no longer be automatically freed when you abandon reference on them, because they reference each other in a cyclic way.
- This is a memory leak.
- You break this by using weak_ptr.
- The "owner" typically use shared_ptr
- and the "owned" use a weak_ptr to its parent,
- and convert it temporarily to shared_ptr when it needs access to its parent.

```
std::shared_ptr<Parent> parentSharedPtr;

std::weak_ptr<Parent> parentWeakPtr =
parentSharedPtr; // automatic conversion to
weak from shared

std::shared_ptr<Parent> tempParentSharedPtr =
parentWeakPtr.lock(); // on the stack, from
the weak ptr

if (!tempParentSharedPtr) {
// yes, it may fail if the parent was freed
since we stored weak_ptr
    }
    else {
        // do stuff
    }

// tempParentSharedPtr is released when it
goes out of scope
```

weak_ptr

```cpp
// In this example we use shared_ptr in cyclically referenced classes. When the classes go out of scope they are NOT destroyed.
//Memory Leak Demo

#include<iostream>
#include<memory>
using namespace std;

class B;

class A
{
public:
    shared_ptr<B>bptr;
    A() {
        cout << "A created" << endl;
    }
    ~A() {
        cout << "A destroyed" << endl;
    }
};

class B
{
public:
    shared_ptr<A>aptr;
    B() {
        cout << "B created" << endl;
    }
    ~B() {
        cout << "B destroyed" << endl;
    }
};

int main()
{
    {
        shared_ptr<A> a = make_shared<A>();
        shared_ptr<B> b = make_shared<B>();
        a->bptr = b;
        b->aptr = a;
    }
}
```

```cpp
// In this example we use weak_ptr to avoid memory leak. When the classes go out of scope they are  destroyed.


#include<iostream>
#include<memory>
using namespace std;

class B;

class A
{
public:
    weak_ptr<B>bptr;
    A() {
        cout << "A created" << endl;
    }
    ~A() {
        cout << "A destroyed" << endl;
    }
};

class B
{
public:
    weak_ptr<A>aptr;
    B() {
        cout << "B created" << endl;
    }
    ~B() {
        cout << "B destroyed" << endl;
    }
};

int main()
{
    {
        shared_ptr<A> a = make_shared<A>();
        shared_ptr<B> b = make_shared<B>();
        a->bptr = b;
        b->aptr = a;
    }

    //it's going out of scope here, and hence destructed
}
```

Component Class

```
namespace GUI
{
    class Component : public sf::Drawable
                    , public sf::Transformable
                    , private sf::NonCopyable
    {
    public:
       typedef std::shared_ptr<Component> Ptr;
    public:
        Component();
        virtual ~Component();
        virtual bool isSelectable() const = 0;
        bool isSelected() const;
        virtual void select();
        virtual void deselect();
        virtual bool isActive() const;
        virtual void activate();
        virtual void deactivate();
        virtual void handleEvent(const sf::Event& event) = 0;
    private:
        bool mIsSelected;
        bool mIsActive;
    };
}
```

Other Classes

- Other classes we define:
  - GUI::Container
  - GUI::Button
  - GUI::Label

- You might recognize some of these as very common GUI types
- They are the most basic components that you will need
  - We can expand the system with more components later

Container Class

```
Container::Container() : mChildren(), mSelectedChild(-1)
{ }

void Container::pack(Component::Ptr component)
{
   mChildren.push_back(component);
   if (!hasSelection() && component->isSelectable())
       select(mChildren.size() - 1);
}

bool Container::isSelectable() const
{
   return false;
}
```

Container Class

```
void Container::handleEvent(const sf::Event& event)
{
      if (hasSelection() && mChildren[mSelectedChild]->isActive())
      {
          mChildren[mSelectedChild]->handleEvent(event);
      }
      else if (event.type == sf::Event::KeyReleased)
      {
            if (event.key.code == sf::Keyboard::W || event.key.code == sf::Keyboard::Up)
            {
               selectPrevious();
            }
            else if (event.key.code == sf::Keyboard::S || event.key.code == sf::Keyboard::Down)
            {
               selectNext();
            }
            else if (event.key.code == sf::Keyboard::Return || event.key.code == sf::Keyboard::Space)
            {
               if (hasSelection())
                        mChildren[mSelectedChild]->activate();
            }
      }
}
```

Container Class

```
void Container::select(std::size_t index)
{
     if (mChildren[index]->isSelectable())
     {
          if (hasSelection())
          mChildren[mSelectedChild]->deselect();
          mChildren[index]->select();
          mSelectedChild = index;
     }
}
void Container::selectNext()
{
     if (!hasSelection())
        return;
     // Search next component that is selectable
     int next = mSelectedChild;
     do
        next = (next + 1) % mChildren.size();
     while (!mChildren[next]->isSelectable());
     // Select that component
     select(next);
}
```

Container Class

```
void Container::selectPrevious()
{
    if (!hasSelection())
      return;
    // Search previous component that is selectable
    int prev = mSelectedChild;
    do
      prev = (prev + mChildren.size() - 1) % mChildren.size();
    while (!mChildren[prev]->isSelectable());
    // Select that component
    select(prev);
}
```

Label Class

```
Label::Label(const std::string& text, const FontHolder& fonts)
: mText(text, fonts.get(Fonts::Label), 16)
{
}

bool Label::isSelectable() const
{
    return false;
}

void Label::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    states.transform *= getTransform();
    target.draw(mText, states);
}

void Label::setText(const std::string& text)
{
    mText.setString(text);
}
```

Button Class

```
Button::Button(const FontHolder& fonts, const TextureHolder& textures)
// ...
{
    mSprite.setTexture(mNormalTexture);
    mText.setPosition(sf::Vector2f(mNormalTexture.getSize() / 2u));
}

bool Button::isSelectable() const
{
    return true;
}

void Button::select()
{
    Component::select();
    mSprite.setTexture(mSelectedTexture);
}

void Button::deselect()
{
    Component::deselect();
    mSprite.setTexture(mNormalTexture);
```

Button Class (cont'd.)

```
void Button::activate()
{
     Component::activate();
     if (mIsToggle)
        mSprite.setTexture(mPressedTexture);
     if (mCallback)
        mCallback();
     if (!mIsToggle)
        deactivate();
}

void Button::deactivate()
{
    Component::deactivate();
     if (mIsToggle)
     {
          if (isSelected())
            mSprite.setTexture(mSelectedTexture);
          else
            mSprite.setTexture(mNormalTexture);
     }
}
```

Updating the Menu

```
MenuState::MenuState(StateStack& stack, Context context)
: State(stack, context), mGUIContainer()
{
    ...
    auto playButton = std::make_shared<GUI::Button>(
      *context.fonts, *context.textures);
    playButton->setPosition(100, 250);
    playButton->setText("Play");
    playButton->setCallback([this] ()
    {
      requestStackPop();
      requestStackPush(States::Game);
    });
    mGUIContainer.pack(playButton);
}
```

Updating the Menu

```
void MenuState::draw()
{
   sf::RenderWindow& window = *getContext().window;
   window.setView(window.getDefaultView());
   window.draw(mBackgroundSprite);
   window.draw(mGUIContainer);
}

bool MenuState::update(sf::Time)
{
   return true;
}

bool MenuState::handleEvent(const sf::Event& event)
{
   mGUIContainer.handleEvent(event);
   return false;
}
```

SettingsState

```
SettingsState::SettingsState(StateStack& stack, Context context)
: State(stack, context)
, mGUIContainer()
{
      mBackgroundSprite.setTexture(
          context.textures->get(Textures::TitleScreen));
      mBindingButtons[Player::MoveLeft] =
          std::make_shared<GUI::Button>(...);
      mBindingLabels[Player::MoveLeft] =
          std::make_shared<GUI::Label>(...);
      ... // More buttons and labels
      updateLabels();
      auto backButton = std::make_shared<GUI::Button>(...);
      backButton->setPosition(100, 375);
      backButton->setText("Back");
      backButton->setCallback([this] ()
      {
          requestStackPop();
      });
      mGUIContainer.pack(mBindingButtons[Player::MoveLeft]);
      mGUIContainer.pack(mBindingLabels[Player::MoveLeft]);
      ...
      mGUIContainer.pack(backButton);
}
```

SettingsState

```
void SettingsState::updateLabels()
{
	Player& player = *getContext().player;
	for (std::size_t i = 0; i < Player::ActionCount; ++i)
	{
		sf::Keyboard::Key key =
		   player.getAssignedKey(static_cast<Player::Action>(i));
		mBindingLabels[i]->setText(toString(key));
	}
}
```

SettingsState

```
bool SettingsState::handleEvent(const sf::Event& event)
{
     bool isKeyBinding = false;
     for (std::size_t action = 0; action < Player::ActionCount; ++action)
     {
          if (mBindingButtons[action]->isActive())
          {
               isKeyBinding = true;
               if (event.type == sf::Event::KeyReleased)
               {
                  getContext().player->assignKey (static_cast<Player::Action>(action),
                    event.key.code);
                  mBindingButtons[action]->deactivate();
               }
               break;
        }
     }
     if (isKeyBinding)
        updateLabels();
     else
        mGUIContainer.handleEvent(event);
     return false;
}
```