Week 14

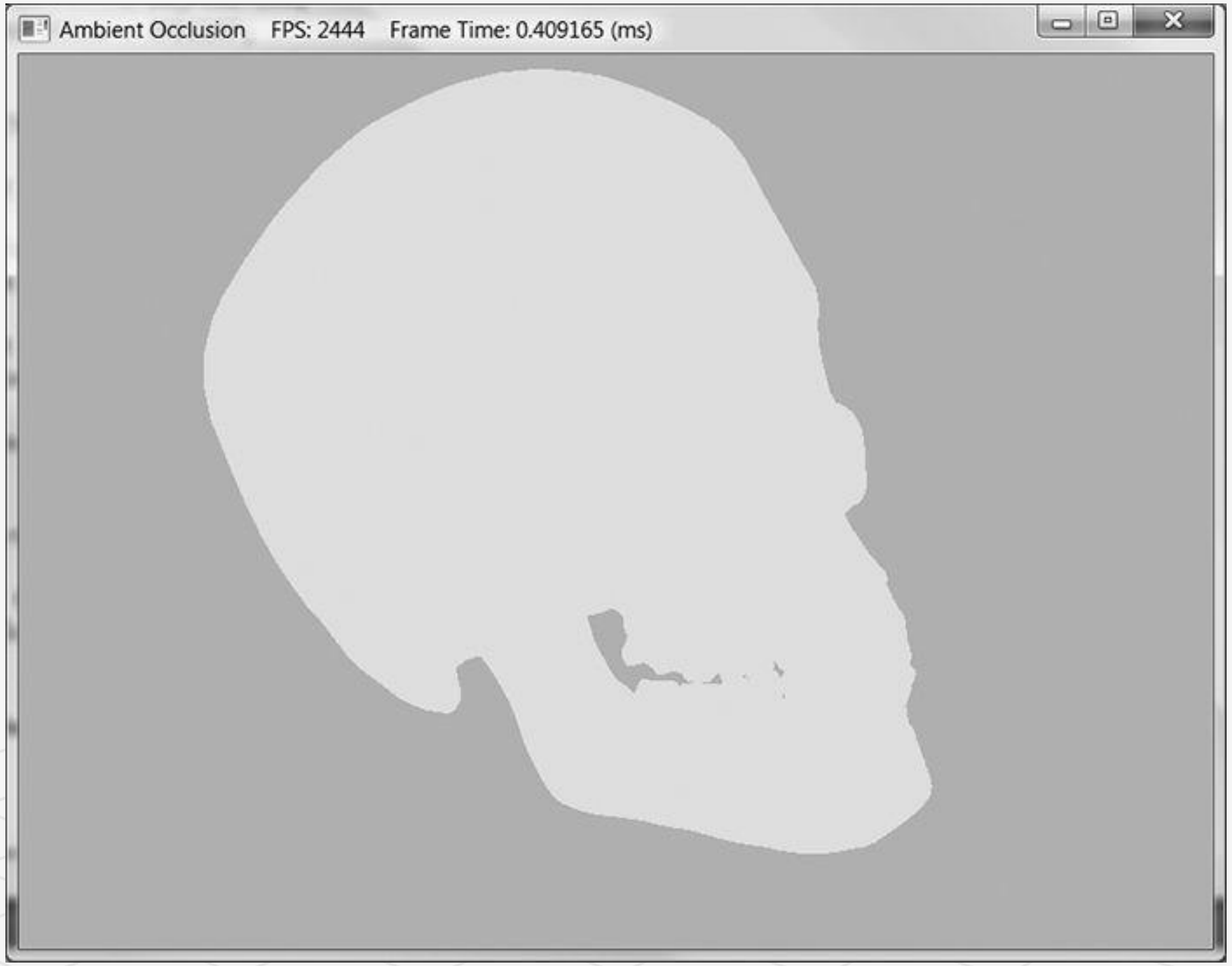
Ambient Occlusion & Quaternions

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.



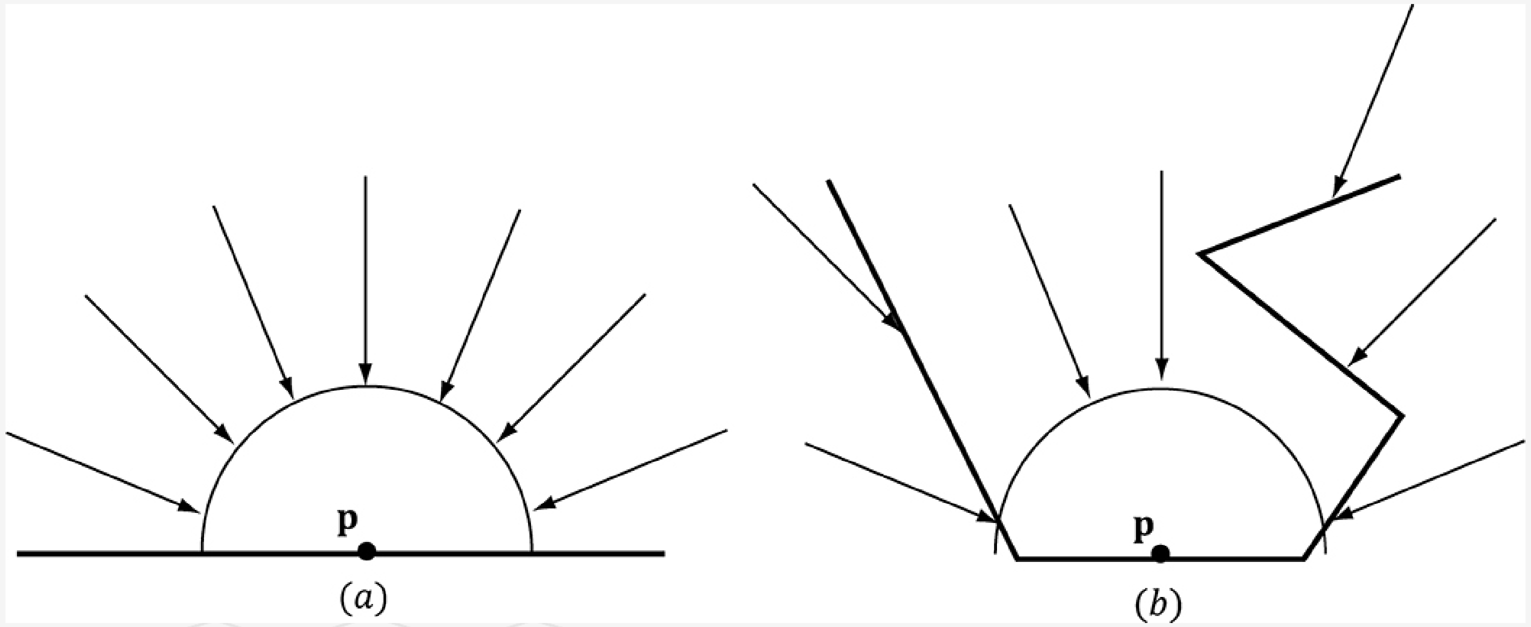
AMBIENT OCCLUSION VIA RAY CASTING

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:

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

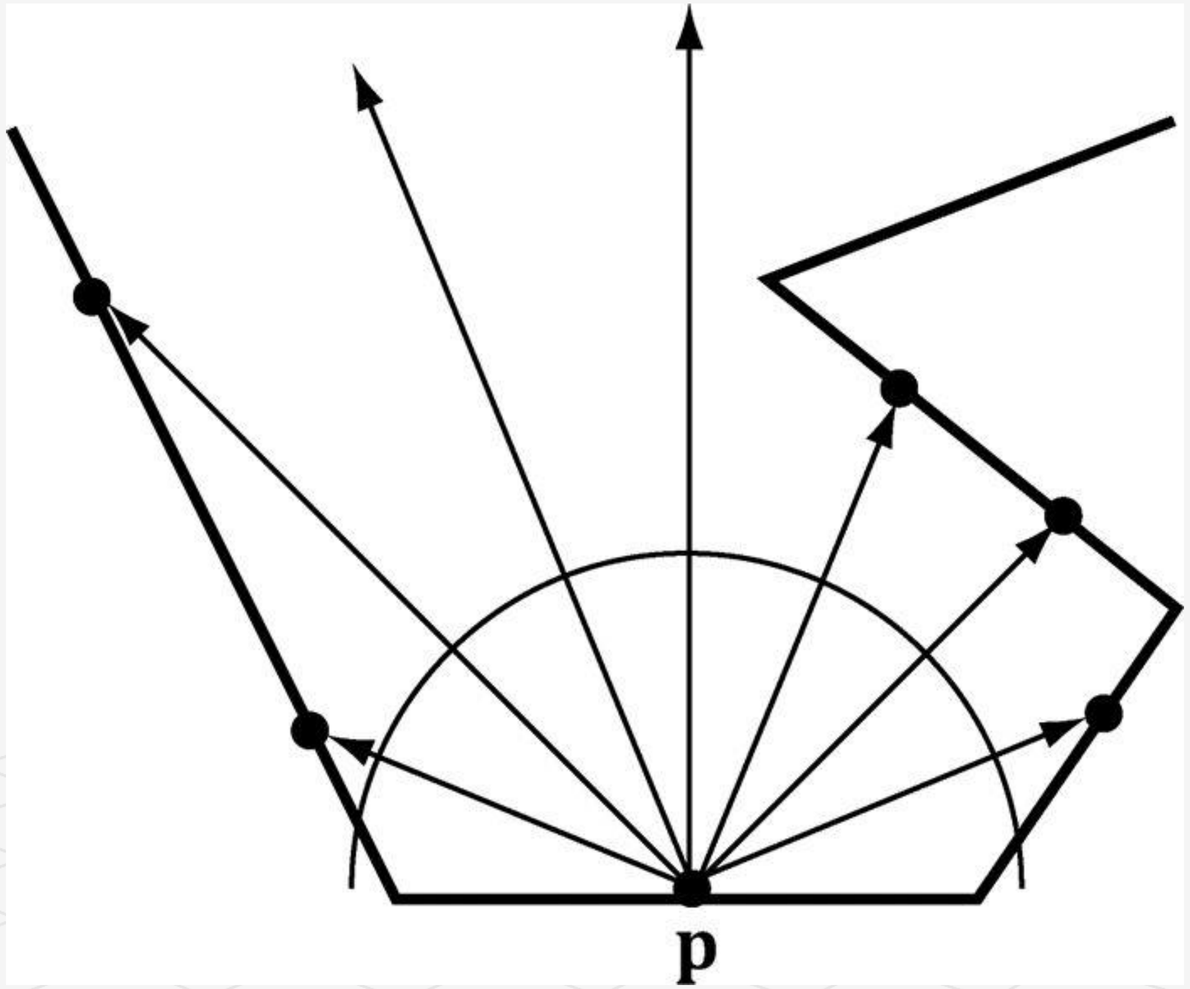
Only rays with an intersection point **q** whose distance from **p** is less than some threshold value *d* should contribute to the occlusion estimate;

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:

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

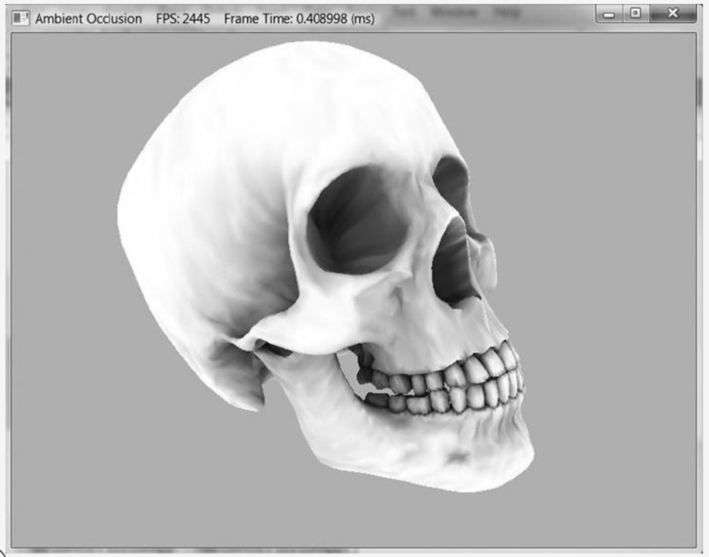


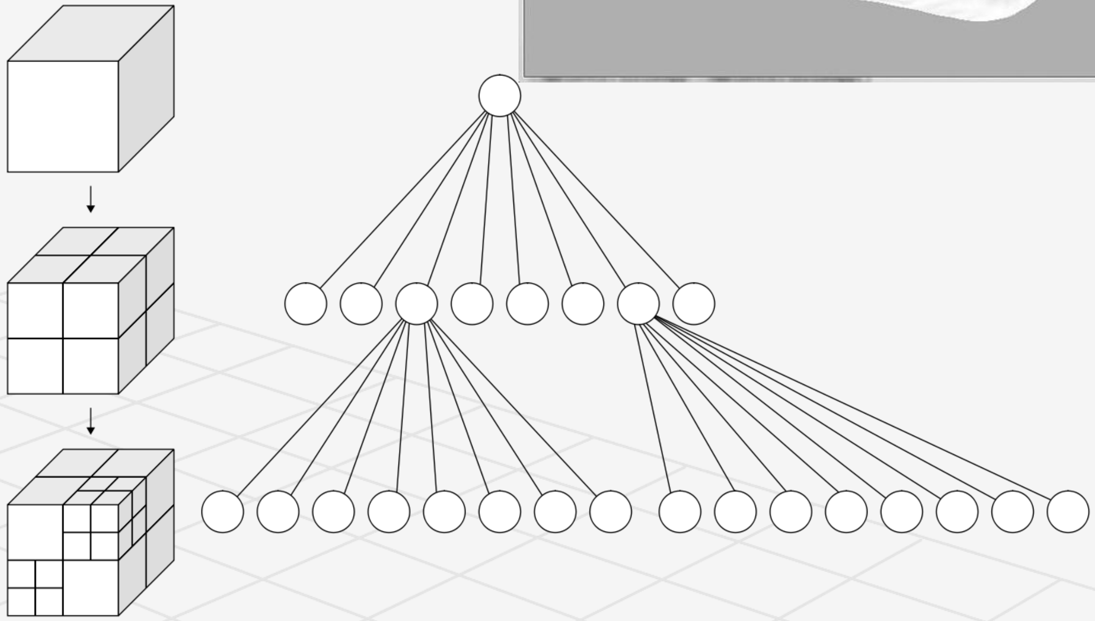
Implementation

The following 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 demo 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.





AmbientOcclusionApp::BuildVertexAmbientOcclusion

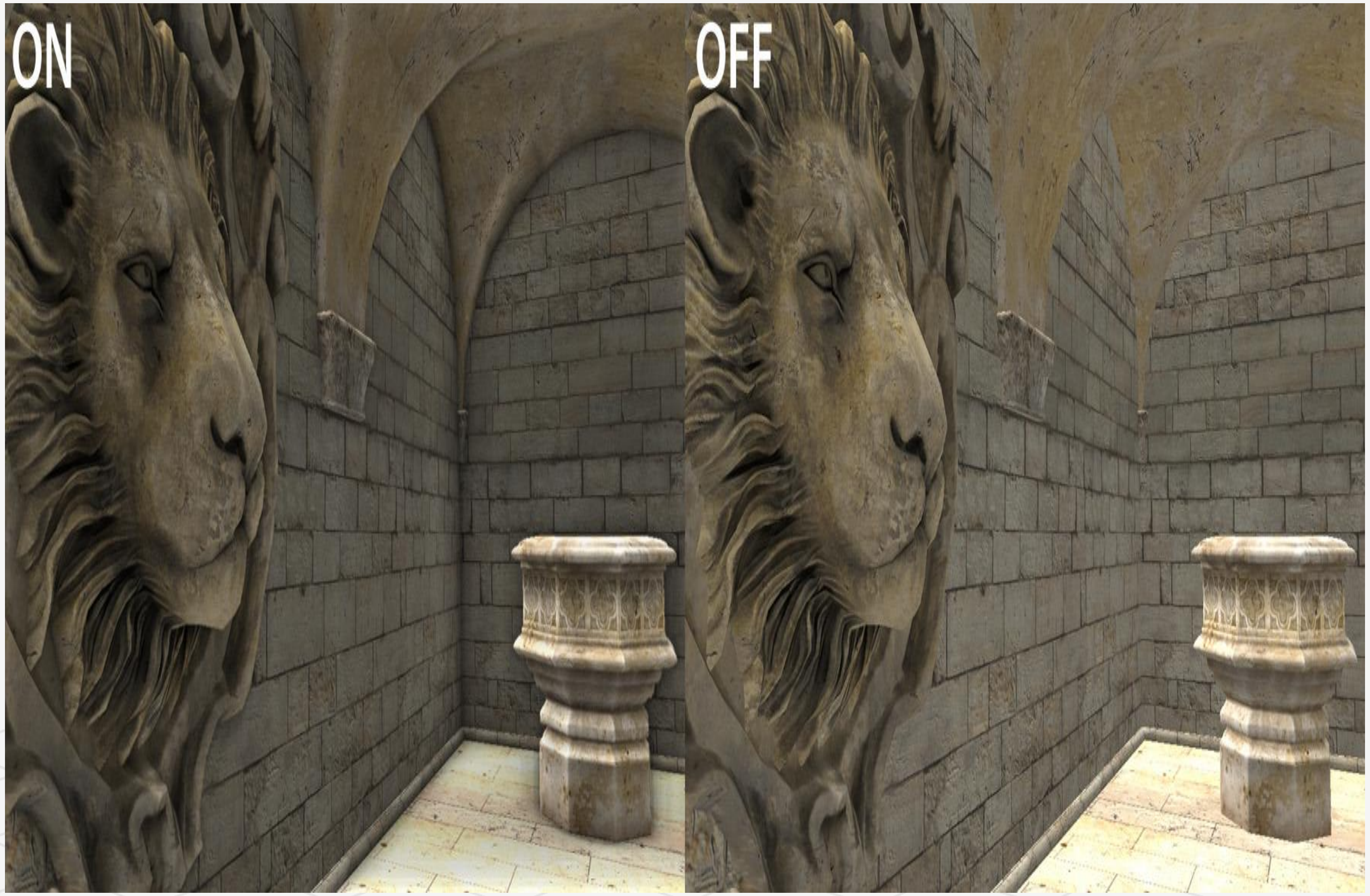
```
void AmbientOcclusionApp::BuildVertexAmbientOcclusion(
std::vector<Vertex::AmbientOcclusion>& vertices,
const std::vector<UINT>& indices)
{
UINT vcount = vertices.size();
UINT tcount = indices.size() / 3;
std::vector<XMFLOAT3> positions(vcount);
for (UINT i = 0; i < vcount; ++i)
positions[i] = vertices[i].Pos;
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.
// TODO: Technically we should not count intersections
// that are far away as occluding the triangle,but  this is OK for demo.
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];
}
}
```

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 Finishing the Calculation

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);
}
```

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:

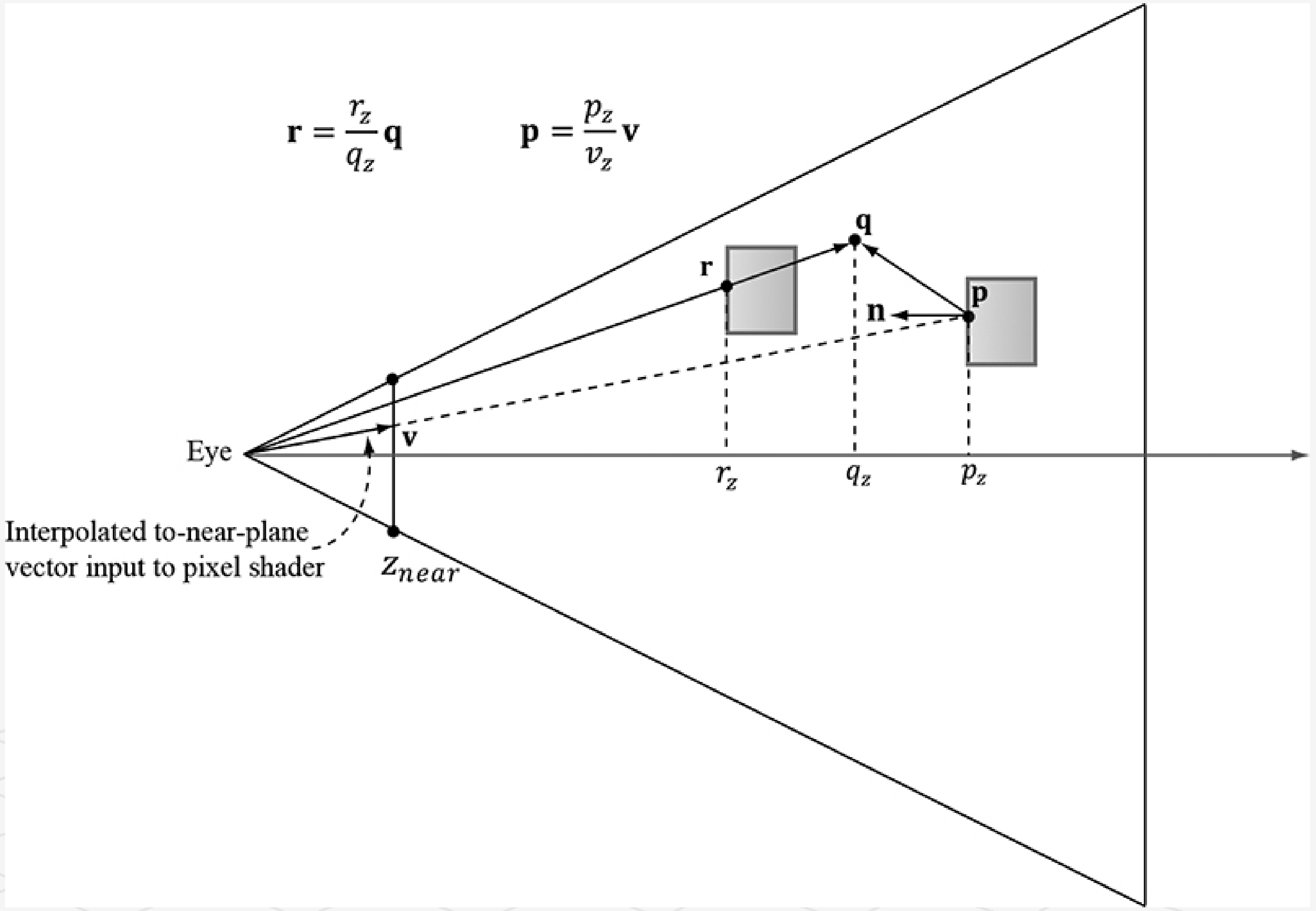
The point **p** corresponds to the current pixel we are processing, and it is reconstructed from the depth value stored in the depth buffer and the vector **v** that passes through the near plane at this pixel.

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 |*pz* − *rz*| 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.



$\mathbf{r} = \frac{r_z}{q_z}\mathbf{q}$
$\mathbf{p} = \frac{p_z}{v_z}\mathbf{v}$
q
r
n
p
Eye
v
r_z
q_z
p_z
Interpolated to-near-plane vector input to pixel shader
z_{near}

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 $t = p_z/v_z$.

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 rz 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 $t = r_z/q_z$.

Therefore $\boldsymbol{r} = r_z/q_z\,\boldsymbol{q}$

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 |*pz* − *rz*|. 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.

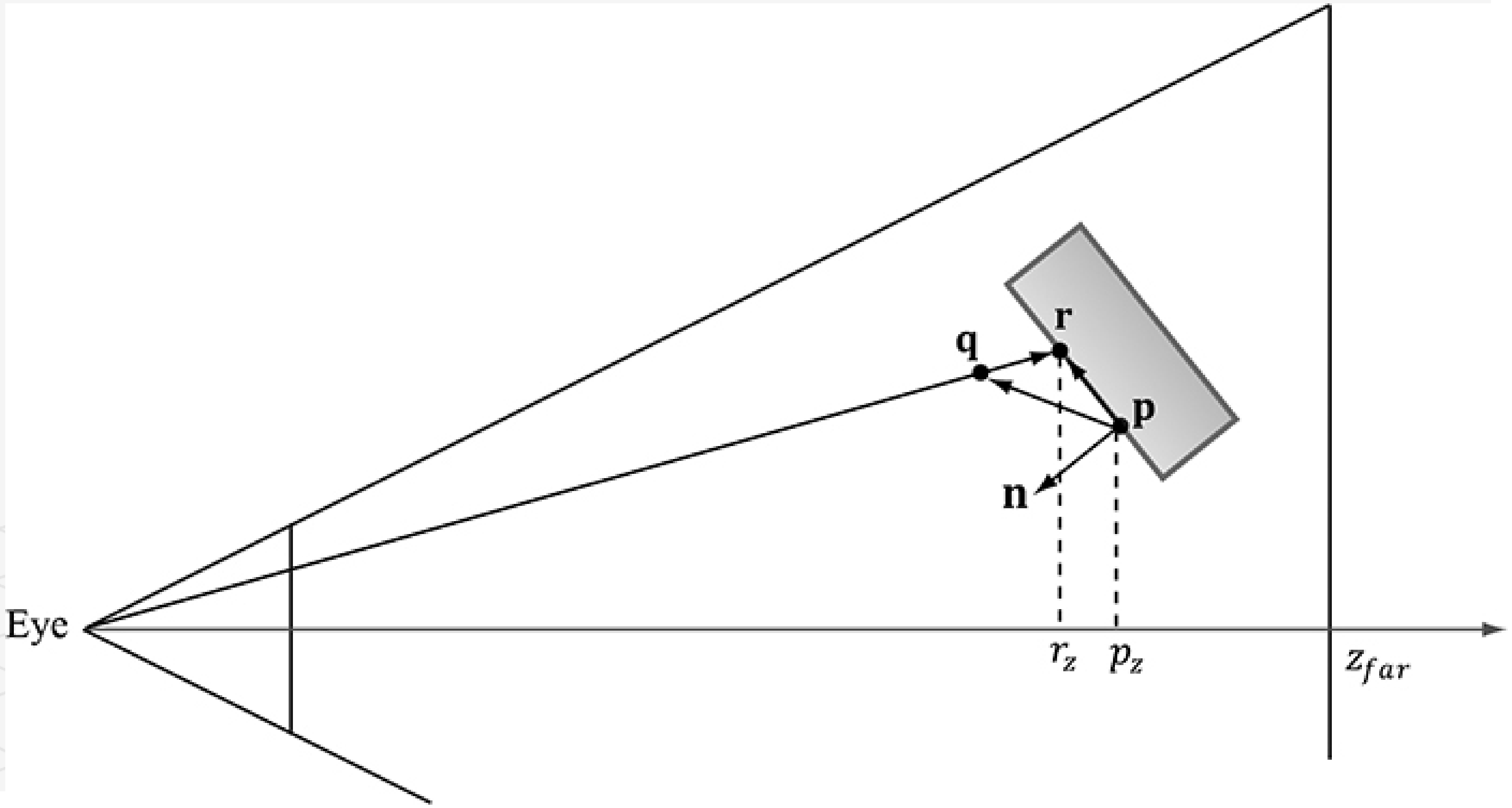
Also, if the distance is very small, then we assume **p** and **q** are on the same plane so **q** cannot occlude **p**.

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

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

This is to prevent self-intersection

If **r** lies on the same plane as **p**, it can pass the first condition that the distance |*pz* − *rz*| 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 $\max(\boldsymbol{n}.\left(r - \frac{p}{||r-p||}\right), 0)$ occlusion by prevents this situation.



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.

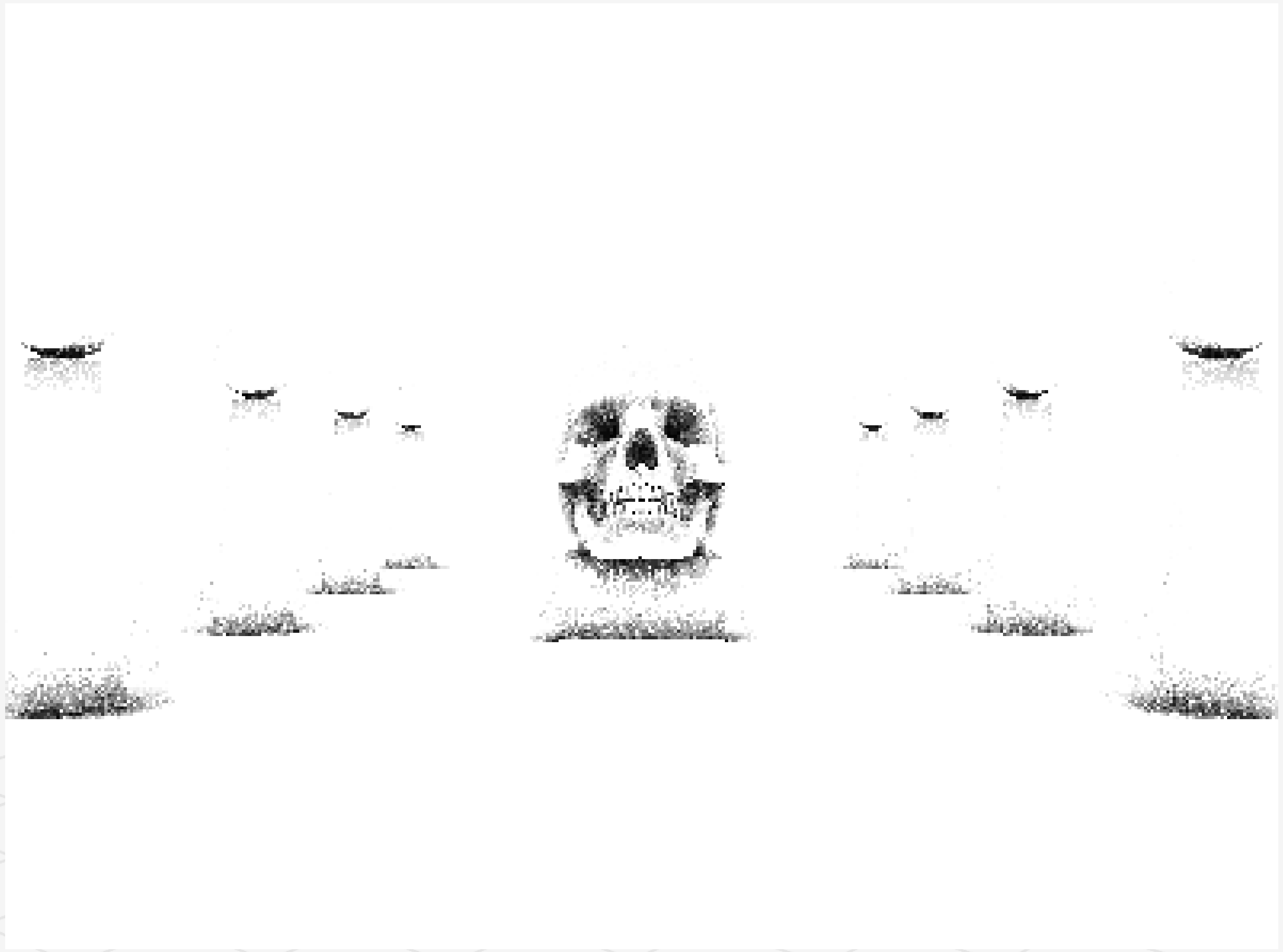
You may also wish to increase the brightness of the ambient map by adding some number to increase the intensity. You can experiment with different contrast/brightness values.

```
// 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

Last figure shows what our ambient occlusion map currently looks like. The noise is due to the fact that we have only taken a few random samples.

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 in Chapter 13, 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, when we render the scene to the back buffer, we bind the ambient map as a shader input. We then generate projective texture coordinates (with respect to the camera), sample the SSAO map, and apply it 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.

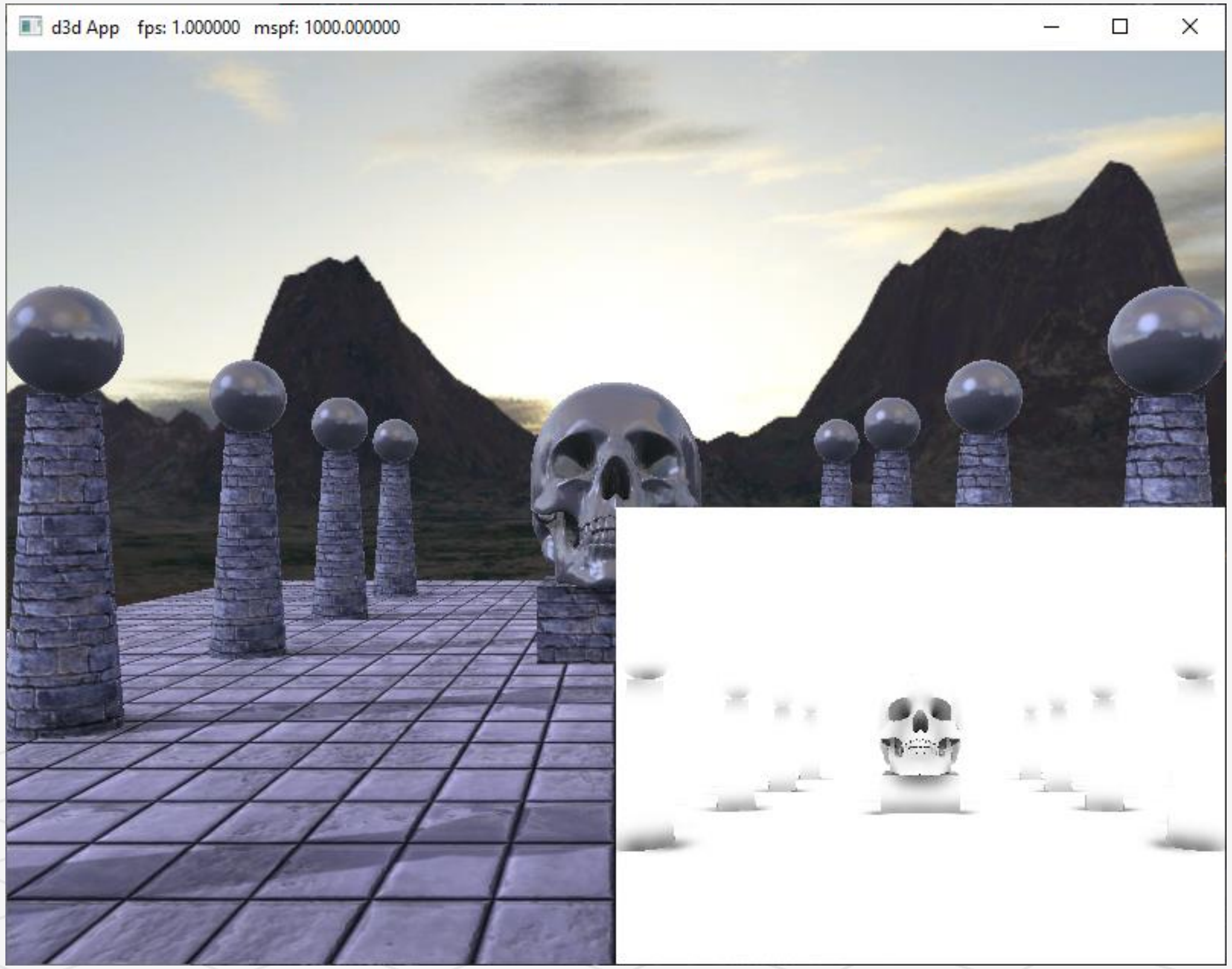
The SSAO can be subtle, and your scene has to reflect enough ambient light so that scaling it by the ambient-access makes enough of a noticeable difference.

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-ssaor2753/

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

Quaternions

Objectives:

1. To review the complex numbers and recall how complex number multiplication performs a rotation in the plane.

2. To obtain an understanding of quaternions and the operations defined on them.

3. To discover how the set of unit quaternions represent 3D rotations.

4. To find out how to convert between the various rotation representations.

5. To learn how to interpolate between unit quaternions, and understand that this is geometrically equivalent to interpolating between 3D orientations.

6. To become familiar with the DirectX Math library's quaternion functions and classes.

REVIEW OF THE COMPLEX NUMBERS

An ordered pair of real numbers **z** = (*a*, *b*) is a complex number.

The first component is called the *real* part and the second component is called the *imaginary* part.

1. $(a, b) = (c, d)$ *if and only if* $a = c$ *and* $b = d$.

2. $(a, b) \pm (c, d) = (a \pm c, b \pm d)$.

3. $(a, b)(c, d) = (ac - bd, ad + bc)$.

4. The *complex conjugate* of a complex number **z** = (*a*, *b*) is denoted by $\bar{Z}$ = (*a*, −*b*).

5. $\frac{(a,b)}{(c,d)} = (\frac{ac+bd}{c^2+d^2}, \frac{bc-ad}{c^2+d^2})$

A simple way to remember the complex division formula is to multiply the numerator and denominator by the conjugate of the denominator so that the denominator becomes a real number:

$$\frac{(a, b)}{(c, d)} = \frac{(a, b)}{(c, d)}\frac{(c, -d)}{(c, -d)} = (\frac{ac + bd}{c^2 + d^2}, \frac{bc - ad}{c^2 + d^2})$$

6. Any real number can be thought of as a complex number with a zero imaginary component: *x* = (*x*, 0);

7. A real number times a complex number is given by *x*(*a*, *b*) = (*x*, 0)(*a*, *b*) = (*xa*, *xb*) = (*a*, *b*)(*x*, 0) = (*a*, *b*)*x*

8. We define the *imaginary unit* $i = (0, 1)$. Using our definition of complex multiplication, $i^2 = (0, 1)(0, 1) = (-1, 0) = -1$, which implies i = $\sqrt{-1}$

9. A complex number (*a*, *b*) can be written in the form *a* + *ib*. We have *a* = (*a*, 0), *b* = (*b*, 0) and *i* = (0, 1), so:

$$a + ib = (a, 0) + (0,1)\,(b, 0) = (a, 0)(0, b) = (a, b)$$

10. $a + ib \pm c + id = (a \pm c) + i(b \pm d)$

11. $(a + ib)(c + id) = (ac - bd) + i(ad + bc)$

12. $\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\,\frac{bc-ad}{c^2+d^2}$ if $(c, d) \neq (0,0)$

13. The complex conjugate of $\boldsymbol{z} = a + ib$ is given by by $\bar{Z}$ = *a* −*ib*

Geometric Interpretation

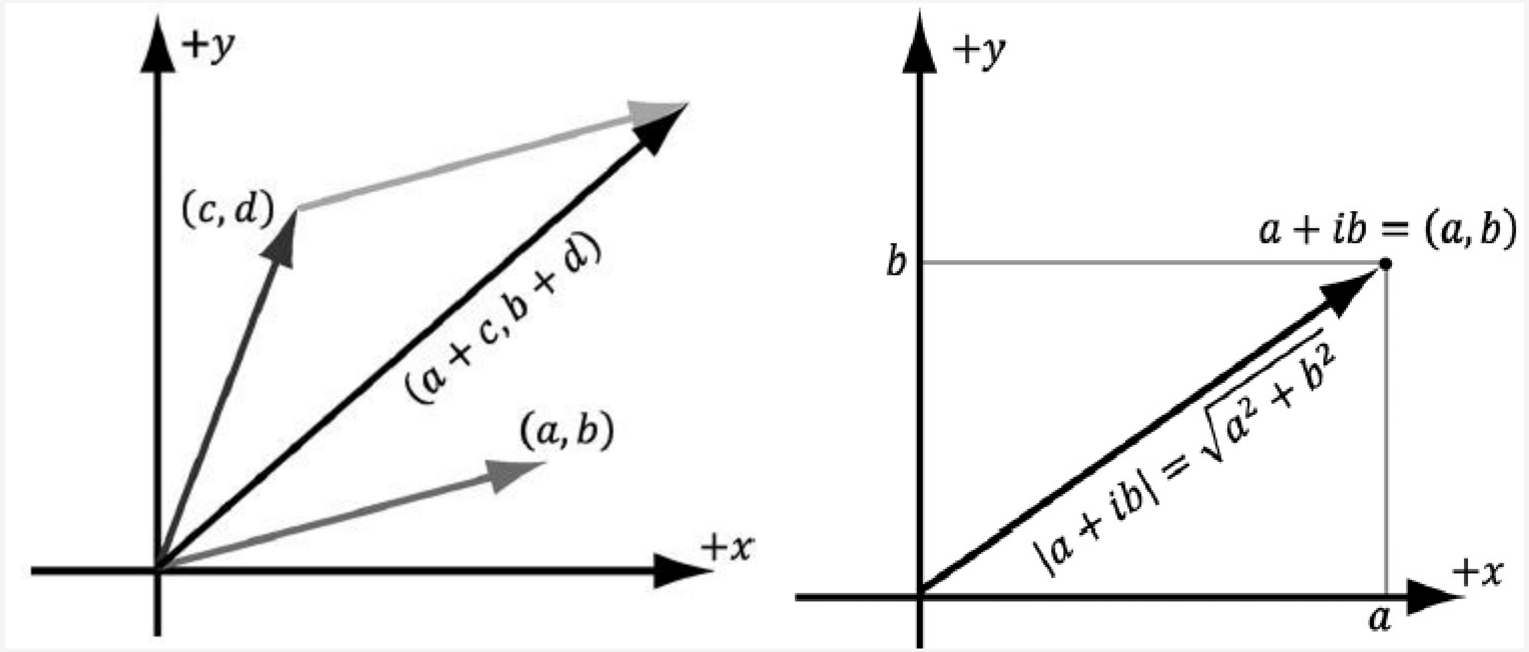
1. The ordered pair form $a + ib = (a, b)$ of a complex number naturally suggests that we think of a complex number geometrically as a 2D point or vector in the complex plane.

2. Complex addition is reminiscent of vector addition in the plane.

3. The *absolute value*, or *magnitude*, of the complex number $a + ib$ is defined as the length of the vector it represents which we know is given by:

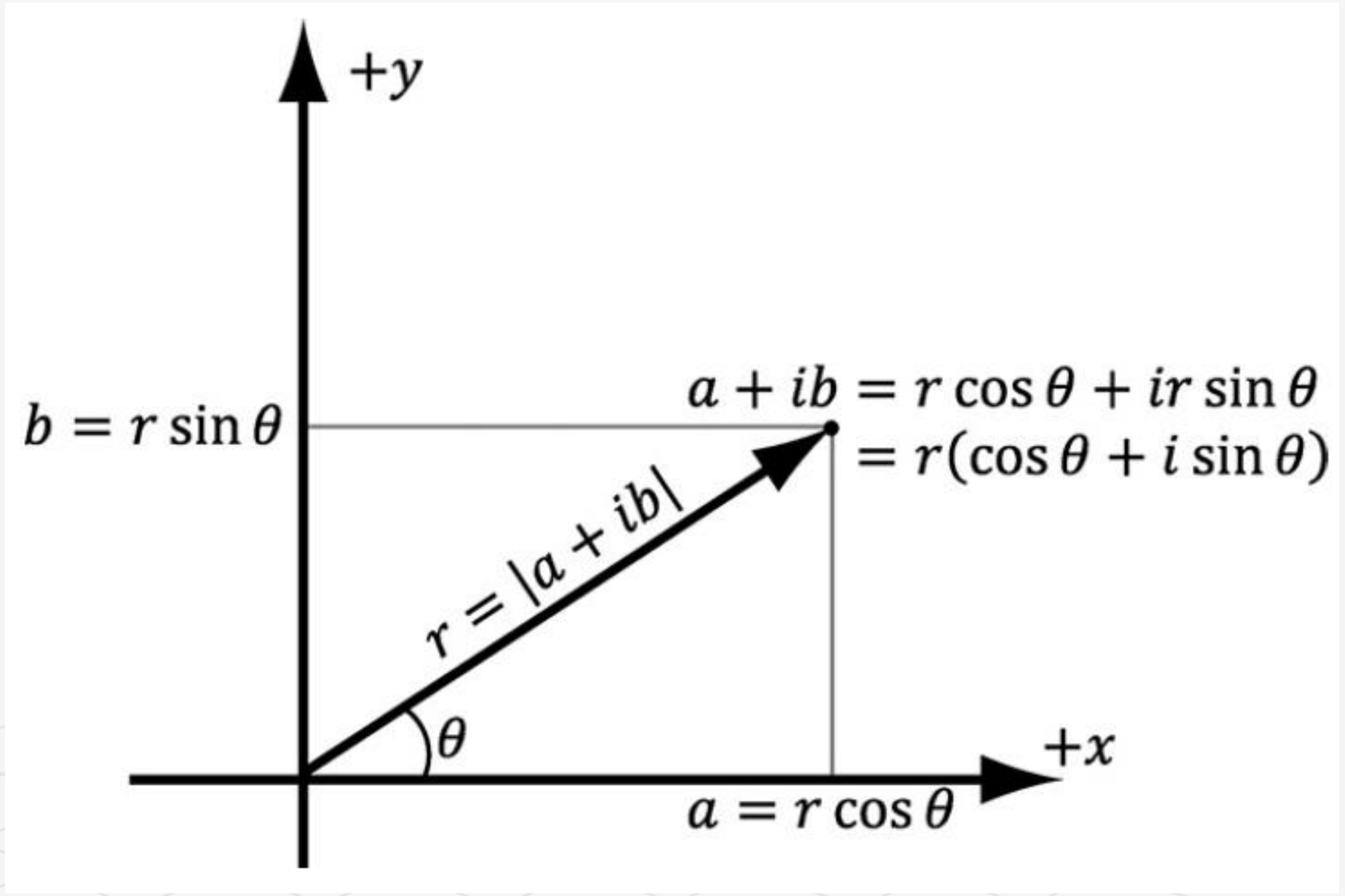
$$| a + ib | = \sqrt{a^2 + b^2}$$

4. A complex number is a *unit complex number* if it has a magnitude of one.



Polar Representation

Because complex numbers can be viewed as just points or vectors in the 2D complex plane, we can just as well express their components using polar coordinates



$+y$
$+x$
$b = r \sin\theta$
$a = r \cos\theta$
$a + ib = r\cos\theta + ir\sin\theta$
$= r(\cos\theta + i\sin\theta)$
$r = |a + ib|$
θ

Rotations

Multiplying a complex number $\mathbf{z}_1$ *(thought of as a 2D vector or point) by a unit complex number* $\mathbf{z}_2$ *results in a rotation of* $\mathbf{z}_1$.

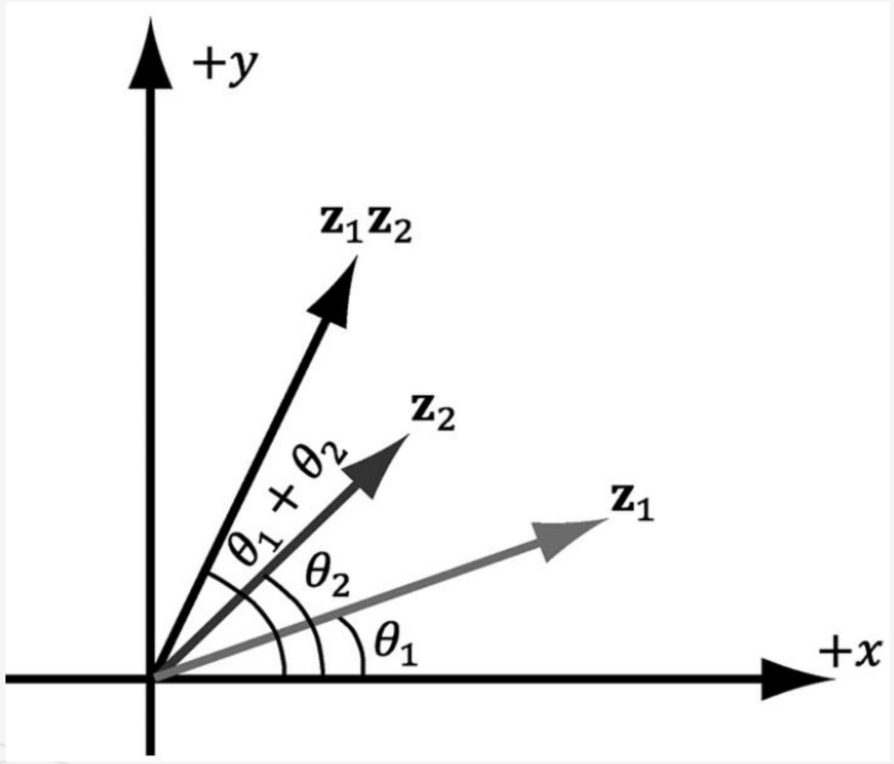
$\boldsymbol{z}_1 = r_1(\cos\theta_1 + isin\theta_1)$,

$\boldsymbol{z}_2 = r_2(\cos\theta_2 + isin\theta_2)$.

Recall
$$\sin(\alpha+\beta) = \sin\alpha\cos\beta + \cos\alpha\sin\beta$$
$$\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$$

The product $\mathbf{z}_1\mathbf{z}_2$ rotates $\mathbf{z}_1$ by the angle θ_2.

$$\begin{aligned}\mathbf{z}_1\mathbf{z}_2 &= r_1r_2(\cos\theta_1\cos\theta_2 - \sin\theta_1\sin\theta_2 + i(\cos\theta_1\sin\theta_2 + \sin\theta_1\cos\theta_2)) \\ &= r_1r_2(\cos(\theta_1+\theta_2) + i\sin(\theta_1+\theta_2))\end{aligned}$$



QUATERNION ALGEBRA

An ordered 4-tuple of real numbers

q = (*x*, *y*, *z*, *w*) = (*q*1, *q*2, *q*3, *q*4) is a quaternion.

This is commonly abbreviated as

q = (**u**, *w*) = (*x*, *y*, *z*, *w*), and we call **u** = (*x*, *y*, *z*) the

imaginary vector part and *w* the real part.

1. (**u**, *a*) = (**v**, *b*) if and only if **u** = **v** and *a* = *b*.

2. (**u**, *a*) ± (**v**, *b*) = (**u** ± **v**, *a* ± *b*).

3. (**u**, *a*)(**v**, *b*) = (*a***v** + *b***u** + **u** × **v**, *ab* − **u** · **v**)

Let **p** = (**u**, *p*4) = (*p*1, *p*2, *p*3, *p*4) and **q** = (**v**, *q*4) = (*q*1, *q*2, *q*3, *q*4). Then **u** × **v** = (*p*2*q*3 – *p*3*q*2, *p*3*q*1 – *p*1*q*3, *p*1*q*2 – *p*2*q*1) and **u·v** = *p*1*q*1 + *p*2*q*2 + *p*3*q*3. Now, in component form, the quaternion product **r** = **pq** takes on the form:

$$
\begin{aligned}
r_1 &= p_4q_1 + q_4p_1 + p_2q_3 - p_3q_2 = q_1p_4 - q_2p_3 + q_3p_2 + q_4p_1 \\
r_2 &= p_4q_2 + q_4p_2 + p_3q_1 - p_1q_3 = q_1p_3 + q_2p_4 - q_3p_1 + q_4p_2 \\
r_3 &= p_4q_3 + q_4p_3 + p_1q_2 - p_2q_1 = -q_1p_2 + q_2p_1 + q_3p_4 + q_4p_3 \\
r_4 &= p_4q_4 - p_1q_1 - p_2q_2 - p_3q_3 = -q_1p_1 - q_2p_2 - q_3p_3 + q_4p_4
\end{aligned}
$$

This can be written as a matrix product:

$$
\mathbf{pq} = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \end{bmatrix}
$$

Special Products

Let **i** = (1, 0, 0, 0), **j** = (0, 1, 0, 0), **k** = (0, 0, 1, 0) be quaternions.

Then we have the special products, some of which are reminiscent of the behavior of the cross product:

$$\begin{gathered}\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1 \\ \mathbf{ij} = \mathbf{k} = -\mathbf{ji} \\ \mathbf{jk} = \mathbf{i} = -\mathbf{kj} \\ \mathbf{ki} = \mathbf{j} = -\mathbf{ik}\end{gathered}$$

Quaternion multiplication is *not* commutative that

$\mathbf{ij} = -\mathbf{ji}$.

Quaternion multiplication is associative, however; this can be seen from the fact that quaternion multiplication can be written using matrix multiplication and matrix multiplication is associative. The quaternion **e** = (0, 0, 0, 1) serves as a multiplicative identity:

$$\mathbf{pe} = \mathbf{ep} = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix}$$

We also have that quaternion multiplication distributes over quaternion addition:

p(**q** + **r**) = **pq** + **pr** and (**q** + **r**)**p** = **qp** + **rp**.

Conversions

We relate real numbers, vectors (or points), and quaternions in the following way:

Let *s* be a real number and let **u** = (*x*, *y*, *z*) be a vector. Then

1. $s = (0, 0, 0, s)$

2. $\mathbf{u} = (x, y, z) = (\mathbf{u}, 0) = (x, y, z, 0)$

In other words, any real number can be thought of as a quaternion with a zero vector part, and any vector can be thought of as a quaternion with zero real part.

The identity quaternion, 1 = (0, 0, 0, 1). A quaternion with zero real part is called a *pure quaternion*.

Observe, using the definition of quaternion multiplication, that a real number times a quaternion is just "scalar multiplication" and it is commutative:

$$s(p_1, p_2, p_3, p_4) = (0,0,0,s)(p_1, p_2, p_3, p_4) = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & s \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{bmatrix} = \begin{bmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{bmatrix}$$

Similarly,

$$(p_1, p_2, p_3, p_4)s = (p_1, p_2, p_3, p_4)(0,0,0,s) = \begin{bmatrix} p_4 & -p_3 & p_2 & p_1 \\ p_3 & p_4 & -p_1 & p_2 \\ -p_2 & p_1 & p_4 & p_3 \\ -p_1 & -p_2 & -p_3 & p_4 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ s \end{bmatrix} = \begin{bmatrix} sp_1 \\ sp_2 \\ sp_3 \\ sp_4 \end{bmatrix}$$

Conjugate and Norm

The conjugate of a quaternion **q** = (*q*1, *q*2, *q*3, *q*4) = (**u**, *q*4) is denoted by **q*** and defined by

q* = *-q1-q2-q3-q4 = (-u,q4)*

In other words, we just negate the imaginary vector part of the quaternion; compare this to the complex number conjugate. The conjugate has the following properties:

$$
\begin{aligned}
&\mathbf{1.}\quad (\mathbf{pq})^* = \mathbf{q}^*\mathbf{p}^* \\
&\mathbf{2.}\quad (\mathbf{p}+\mathbf{q})^* = \mathbf{p}^* + \mathbf{q}^* \\
&\mathbf{3.}\quad (\mathbf{q}^*)^* = \mathbf{q} \\
&\mathbf{4.}\quad (s\mathbf{q})^* = s\mathbf{q}^* \text{ for } s \in \mathbb{R} \\
&\mathbf{5.}\quad \mathbf{q}+\mathbf{q}^* = (\mathbf{u},q_4)+(-\mathbf{u},q_4) = (0,2q_4) = 2q_4 \\
&\mathbf{6.}\quad \mathbf{qq}^* = \mathbf{q}^*\mathbf{q} = q_1^2+q_2^2+q_3^2+q_4^2 = \|\mathbf{u}\|^2+q_4^2
\end{aligned}
$$

The *norm* (or *magnitude*) of a quaternion is defined by: $\|\mathbf{q}\| = \sqrt{\mathbf{qq}^*} = \sqrt{q_1^2+q_2^2+q_3^2+q_4^2} = \sqrt{\|\mathbf{u}\|^2+q_4^2}$

We say that a quaternion is a *unit quaternion* if it has a norm of one. The norm has the

following properties: 1. ||q*|| = ||q|| 2. ||pq|| = ||p||||q||

In particular, property 2 tells us that the product of two unit quaternions is a unit

quaternion; also if ||**p||** = 1, then ||**pq**|| = ||**q**||. The conjugate and norm properties can be derived straightforwardly from the definitions. For example,

$$(\mathbf{q}^*)^* = (-\mathbf{u},q_4)^* = (\mathbf{u},q_4) = \mathbf{q}$$

$$\|\mathbf{q}^*\| = \|(-\mathbf{u},q_4)\| = \sqrt{\|-\mathbf{u}\|^2+q_4^2} = \sqrt{\|\mathbf{u}\|^2+q_4^2} = \|\mathbf{q}\|$$

$$
\begin{aligned}
\|\mathbf{pq}\|^2 &= (\mathbf{pq})(\mathbf{pq})^* \\
&= \mathbf{pqq}^*\mathbf{p}^* \\
&= \mathbf{p}\|\mathbf{q}\|^2\mathbf{p}^* \\
&= \mathbf{pp}^*\|\mathbf{q}\|^2 \\
&= \|\mathbf{p}\|^2\|\mathbf{q}\|^2
\end{aligned}
$$

Inverses

As with matrices, quaternion multiplication is not commutative, so we cannot define a division operator.

However, every nonzero quaternion has an inverse. (The zero quaternion has zeros for all its components.) Let **q** = (*q*1, *q*2, *q*3, *q*4) = (**u**, *q*4) be a nonzero quaternion, then the inverse is denoted by **q**−1 and given by:

$$q - {}^{1} = \frac{q*}{||q||^2}$$

$$(q - {}^{1})^{-1} = q$$

$$(pq)^{-1} = q^{-1} p^{-1}$$

Polar Representation

If **q** = (q_1, q_2, q_3, q_4) = (**u**, q_4) is a unit quaternion, then $||q||^2 = ||u||^2 + {q_4}^2 = 1$

Figure shows there exists an angle θ∈[0, π] such that q_4 = cosθ.

Employing the trigonometric identity

$\sin^2\theta + \cos^2\theta = 1$, we have that

$\sin^2\theta = 1 - \cos^2\theta = 1 - {q_4}^2 = ||u||^2$

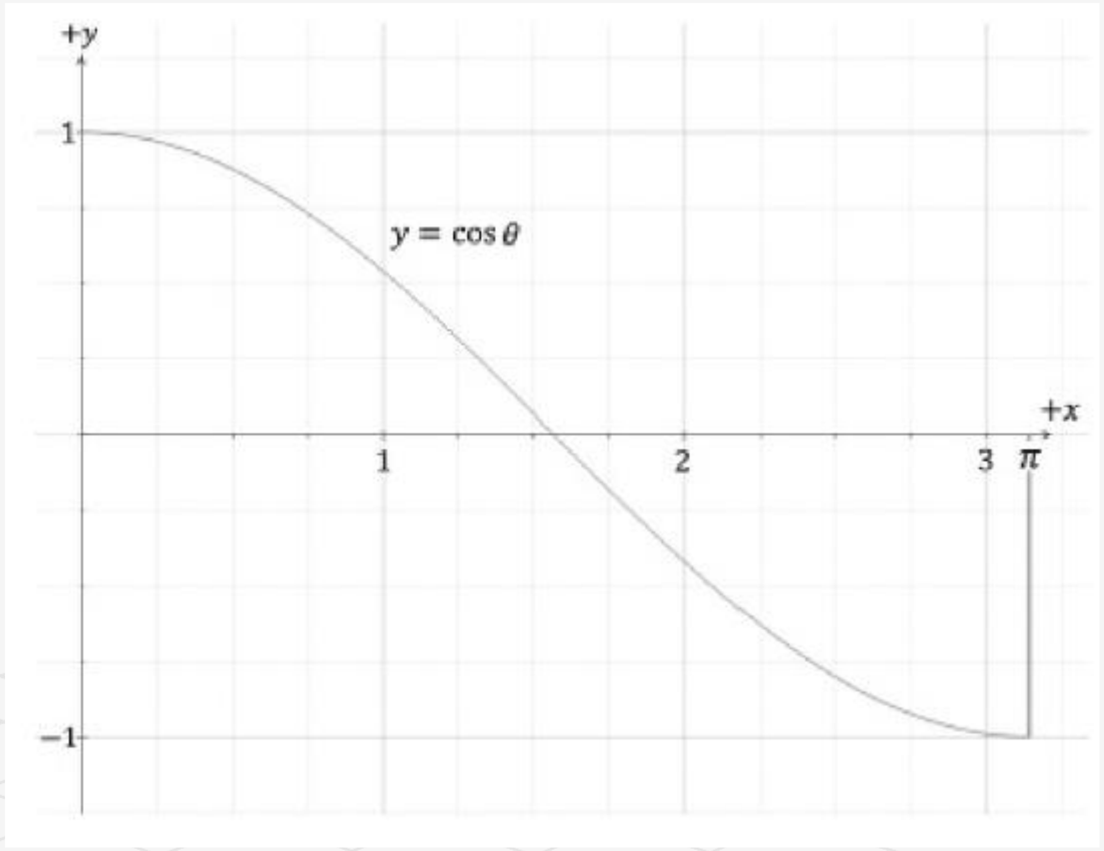
Now label the unit vector in the same direction as **u** by **n**: n = u/||u|| = u/sinθ

Hence, **u** = sinθ**n** and, the unit quaternion **q** = (**u**, q4) in the following *polar representation* where **n** is a unit vector:

$$q = (\sin\theta \boldsymbol{n}, cos\theta)\ for\ \theta \in [0, \pi]$$

Substituting -θ to θ is equivalent to negating the vector part of the quaternion:

$$(n\sin(-\theta), \cos(-\theta)) = (-n\sin\theta, cos\theta) = p^*$$



UNIT QUATERNIONS AND ROTATIONS

Let **q** = (**u**, *w*) be a unit quaternion and let **v** be a 3D point or vector. Then we can think

of **v** as the pure quaternion **p** = (**v**, 0). Also recall that since **q** is a unit quaternion, we have that q−1 = q*.

Recall the formula for quaternion multiplication:

$$(m, a)(n, b) = (an + bm + m\ x\ n, ab - m.n)$$

The quaternion rotation operator *R*q (v) = qvq−1 rotates a vector (or point) **v** about the axis **n** by an angle 2θ.

$$\begin{aligned} R_{\mathbf{q}}(\mathbf{v}) &= \mathbf{q}\mathbf{v}\mathbf{q}^{-1} \\ &= \mathbf{q}\mathbf{v}\mathbf{q}^{*} \\ &= \cos(2\theta)\mathbf{v} + (1-\cos(2\theta))(\mathbf{n}\cdot\mathbf{v})\mathbf{n} + \sin(2\theta)(\mathbf{n}\times\mathbf{v}) \end{aligned}$$

So suppose you are given an axis **n** and angle θ to rotate about the axis **n**. You

construct the corresponding rotation quaternion by:

$$\mathbf{q} = \left(\sin\left(\frac{\theta}{2}\right)\mathbf{n}, \cos\left(\frac{\theta}{2}\right) \right)$$

Then apply the formula *R***q**(**v**). The division by 2 is to compensate for the 2θ because

we want to rotate by the angle θ, not 2θ.

Quaternion Rotation Operator to Matrix

Let **q** = (**u**, *w*) = (*q*1, *q*2, *q*3, *q*4) be a unit quaternion.

$$R_{\mathbf{q}}(\mathbf{v}) = \mathbf{v}\mathbf{Q} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4 \\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4 \\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

Matrix to Quaternion Rotation Operator

Given the rotation matrix

$$\mathbf{R} = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix}$$

, we want to find the quaternion **q** = (*q*1, *q*2, *q*3, *q*4) such that if we build the matrix **Q**

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{bmatrix} = \begin{bmatrix} 1-2q_2^2-2q_3^2 & 2q_1q_2+2q_3q_4 & 2q_1q_3-2q_2q_4 \\ 2q_1q_2-2q_3q_4 & 1-2q_1^2-2q_3^2 & 2q_2q_3+2q_1q_4 \\ 2q_1q_3+2q_2q_4 & 2q_2q_3-2q_1q_4 & 1-2q_1^2-2q_2^2 \end{bmatrix}$$

We start by summing the diagonal elements (which is called the *trace* of a matrix): from **q** we get **R**.

$$\begin{aligned} \text{trace}(\mathbf{R}) &= R_{11}+R_{22}+R_{33} \\ &= 1-2q_2^2-2q_3^2+1-2q_1^2-2q_3^2+1-2q_1^2-2q_2^2 \\ &= 3-4q_1^2-4q_2^2-4q_3^2 \\ &= 3-4(q_1^2+q_2^2+q_3^2) \\ &= 3-4(1-q_4^2) \\ &= -1+4q_4^2 \\ \therefore q_4 &= \frac{\sqrt{\text{trace}(\mathbf{R})+1}}{2} \end{aligned}$$

Now we combine diagonally opposite elements to solve for q_1, q_2, q_3 (because we eliminate terms):

$$\begin{aligned} R_{23}-R_{32} &= 2q_2q_3+2q_1q_4-2q_2q_3+2q_1q_4 \\ &= 4q_1q_4 \\ \therefore \quad &= \underline{R_{23} \quad R_{32}} \end{aligned}$$

$$\begin{aligned} R_{31}-R_{13} &= 2q_1q_3+2q_2q_4-2q_1q_3+2q_2q_4 \\ &= 4q_2q_4 \\ \therefore q_2 &= \frac{R_{31}-R_{13}}{4q_4} \end{aligned}$$

$$\begin{aligned} R_{12}-R_{21} &= 2q_1q_2+2q_3q_4-2q_1q_2+2q_3q_4 \\ &= 4q_3q_4 \\ \therefore q_3 &= \frac{R_{12}-R_{21}}{4q_4} \end{aligned}$$

Composition

Suppose **p** and **q** are unit quaternions with corresponding rotational operators given by *R***p** and *R***q**, respectively. Letting

$v' = R_p(v)$

the composition is given by:

$$R_{\mathbf{q}}(R_{\mathbf{p}}(\mathbf{v})) = R_{\mathbf{q}}(\mathbf{v}') = \mathbf{q}\mathbf{v}'\mathbf{q}^{-1} = \mathbf{q}(\mathbf{p}\mathbf{v}\mathbf{p}^{-1})\mathbf{q}^{-1} = (\mathbf{q}\mathbf{p})\mathbf{v}(\mathbf{p}^{-1}\mathbf{q}^{-1}) = (\mathbf{q}\mathbf{p})\mathbf{v}(\mathbf{q}\mathbf{p})^{-1}$$

Because **p** and **q** are both unit quaternions, the product **pq** is also a unit quaternion since ||**pq**|| = ||**p**||||**q**|| = 1; thus, the quaternion product **pq** also represents a rotation

QUATERNION INTERPOLATION

Since quaternions are 4-tuples of real numbers, geometrically, we can visualize them as 4D vectors.

Unit quaternions are 4D unit vectors that lie on the 4D unit sphere.

With the exception of the cross product (which is only defined for 3D vectors), our vector math generalizes to 4-space—and even *n*-space.

Specifically, the dot product holds for quaternions. Let **p** = (**u**, *s*) and **q** = (**v**, *t*), then:

$p.q = u.v + st = ||p|| ||q|| cos\theta$

Where θ is the angle between the quaternions. If the quaternions **p** and **q** are unit length, then **p**·**q** = cosθ

The dot product allows us to talk about the angle between two quaternions, as a measure of how "close" they are to each other on the unit sphere.

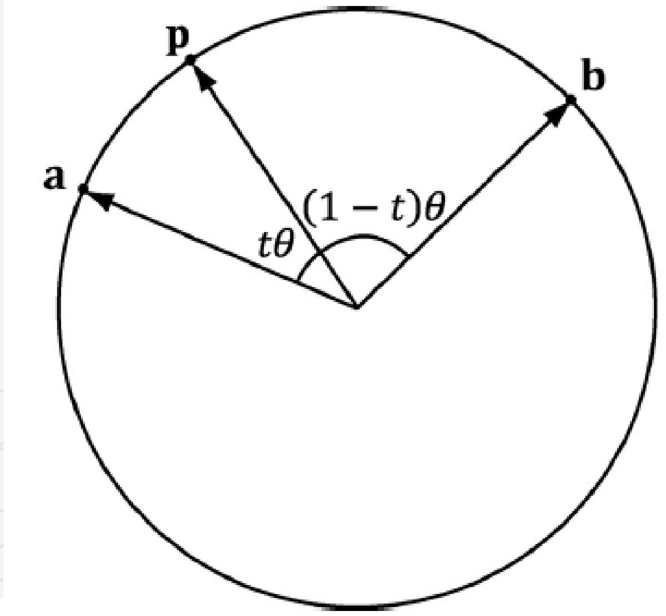
For the purposes of animation, we want to interpolate from one orientation to another orientation. To interpolate quaternions, we want to interpolate on the arc of the unit sphere so that our interpolated quaternion is also a unit quaternion.

The following figure shows interpolating along the 4D unit sphere from **a** to **b** by an angle *t*θ. The angle between **a** and **b** is θ, the angle between **a** and **p** is *t*θ, and the angle between **p** and **b** is (1 − *t*)θ.

Thus we define the spherical interpolation formula:

$$\text{slerp}(\mathbf{a},\mathbf{b},t) = \frac{\sin((1-t)\theta)\mathbf{a} + \sin(t\theta)\mathbf{b}}{\sin\theta} \quad \text{for} \quad t \in [0,1]$$

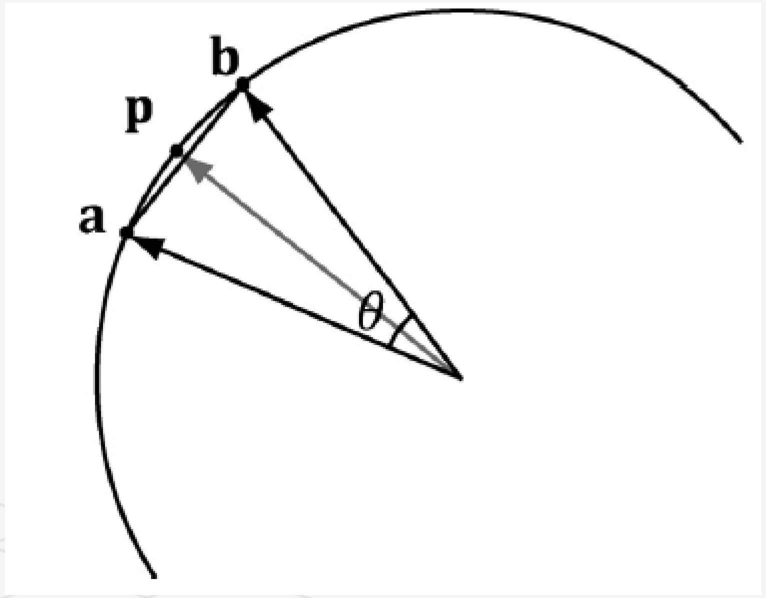
Thinking of unit quaternions as 4D unit vectors allows us to solve for the angle between the quaternions:
θ = arccos(a · b).
If θ, the angle between **a** and **b** is near zero, sinθ is near zero, and the division can cause problems due to finite numerical precision. In this case, perform linear interpolation between the quaternions and normalize the result, which is actually a good approximation for small θ



QUATERNION INTERPOLATION

For small angles θ between **a** and **b**, linear interpolation is a good approximation for spherical interpolation.

However, when using linear interpolation,the interpolated quaternion no longer lies on the unit sphere, so you must normalizethe result to project it back on to the unit sphere.



QUATERNION INTERPOLATION

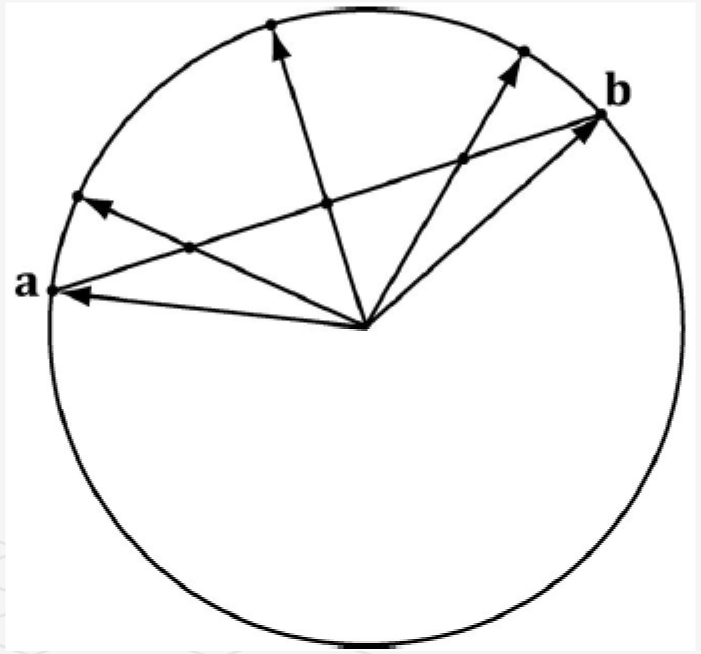
Notice that that linear interpolation followed by projecting the interpolated quaternion back on to the unit sphere results in a nonlinear rate of rotation.

we used linear interpolation for large angles, the speed of rotation will speed up and slow down.

This effect is often undesirable, and one reason why spherical interpolation is preferred (which rotates at a constant speed).

Linear interpolation results in nonlinear interpolation over the unit sphere after normalization.

This means the rotation speeds up and slows down as it interpolates, rather than moving at a constant speed.

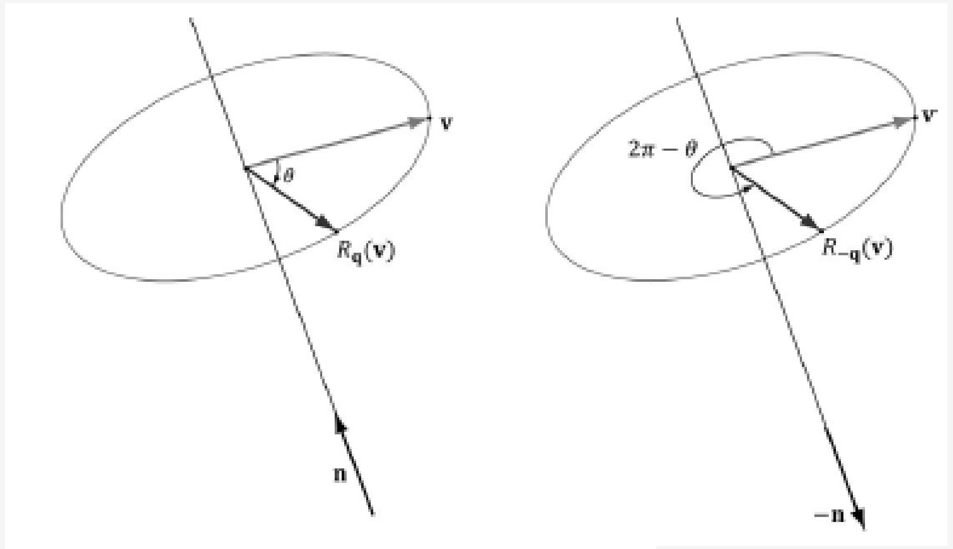


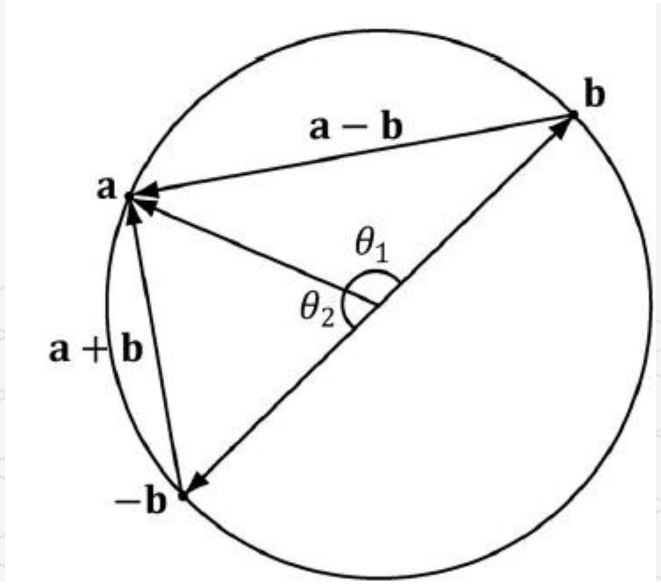
QUATERNION INTERPOLATION

*R*q rotates θ about the axis **n**, and *R*−q rotates 2π − θ about the axis **−n**.

Interpolating from **a** to **b** results in interpolating over the larger arc θ1 on the 4D unit sphere, whereas interpolating from **a** to −**b** results in interpolating over the shorter arc θ2 on the 4D unit sphere.

We want to choose the shortest arc on the 4D unit sphere.





LerpAndNormalize and Slerp functions

```
// Linear interpolation (for small theta).
public static Quaternion
LerpAndNormalize(Quaternion p, Quaternion q,
float s)
{
// Normalize to make sure it is a unit
quaternion.
return Normalize((1.0f - s) * p + s * q);
}
```

```
public static Quaternion Slerp(Quaternion p,
Quaternion q, float s)
{
// Recall that q and -q represent the same orientation, but
// interpolating between the two is different: One will take the
// shortest arc and one will take the long arc. To find
// the shortest arc, compare the magnitude of p-q with the
// magnitude p-(-q) = p+q.
if (LengthSq(p - q) > LengthSq(p + q))
q = -q;
float cosPhi = DotP(p, q);
// For very small angles, use linear interpolation.
if (cosPhi > (1.0f - 0.001))
return LerpAndNormalize(p, q, s);
// Find the angle between the two quaternions.
float phi = (float)Math.Acos(cosPhi);
float sinPhi = (float)Math.Sin(phi);
// Interpolate along the arc formed by the
intersection of the 4D
// unit sphere and the plane passing through p, q,
and the origin of
// the unit sphere.
return ((float)Math.Sin(phi * (1.0 - s)) / sinPhi) * p +
((float)Math.Sin(phi * s) / sinPhi) * q;
}
```

DIRECTX MATH QUATERNION FUNCTIONS

The DirectX math library supports quaternions. Because the "data" of a quaternion is four real numbers, DirectX math uses the XMVECTOR type for storing quaternions. Then some of the common quaternion functions defined are:

```
// Returns the quaternion dot product Q1·Q2.
XMVECTOR XMQuaternionDot(XMVECTOR Q1, XMVECTOR Q2);
// Returns the identity quaternion (0, 0, 0, 1).
XMVECTOR XMQuaternionIdentity();
// Returns the conjugate of the quaternion Q.
XMVECTOR XMQuaternionConjugate(XMVECTOR Q);
// Returns the norm of the quaternion Q.
XMVECTOR XMQuaternionLength(XMVECTOR Q);
// Normalizes a quaternion by treating it as a 4D vector.
XMVECTOR XMQuaternionNormalize(XMVECTOR Q);
// Computes the quaternion product Q1Q2.
XMVECTOR XMQuaternionMultiply(XMVECTOR Q1, XMVECTOR Q2);
// Returns a quaternions from axis-angle rotation representation.
XMVECTOR XMQuaternionRotationAxis(XMVECTOR Axis, FLOAT Angle);
// Returns a quaternions from axis-angle rotation representation, where the axis
// vector is normalized—this is faster than XMQuaternionRotationAxis.
XMVECTOR XMQuaternionRotationNormal(XMVECTOR NormalAxis,FLOAT Angle);
// Returns a quaternion from a rotation matrix.
XMVECTOR XMQuaternionRotationMatrix(XMMATRIX M);
// Returns a rotation matrix from a unit quaternion.
XMMATRIX XMMatrixRotationQuaternion(XMVECTOR Quaternion);
// Extracts the axis and angle rotation representation from the quaternion Q.
VOID XMQuaternionToAxisAngle(XMVECTOR *pAxis, FLOAT *pAngle, XMVECTOR Q);
// Returns slerp(Q1, Q2, t)
XMVECTOR XMQuaternionSlerp(XMVECTOR Q0, XMVECTOR Q1, FLOAT t);
```

ROTATION DEMO

We animate a skull mesh around a simple scene.

The position, orientation, and scale of the mesh are animated.

We use quaternions to represent the orientation of the skull, and use slerp to interpolate between orientations.

We use linear interpolation to interpolate between position and scale.

A common form of animation is called key frame animation.

A *key frame* specifies the position, orientation, and scale of an object at an instance in time. In our demo (in *AnimationHelper.h/.cpp*), we define the following key frame structure:

```
Keyframe::Keyframe()
: TimePos(0.0f),
Translation(0.0f, 0.0f, 0.0f),
Scale(1.0f, 1.0f, 1.0f),
RotationQuat(0.0f, 0.0f, 0.0f, 1.0f)
{
}

Keyframe::~Keyframe()
{
}

float BoneAnimation::GetStartTime()const
{
// Keyframes are sorted by time, so first keyframe gives start time.
return Keyframes.front().TimePos;
}

float BoneAnimation::GetEndTime()const
{
// Keyframes are sorted by time, so last keyframe gives end time.
float f = Keyframes.back().TimePos;

return f;
}

void BoneAnimation::Interpolate(float t, XMFLOAT4X4& M)const
{
```

BoneAnimation::Interpolate

We now have a list of key frames, which define the rough overall look of the animation.

How will the animation look at time between the key frames?

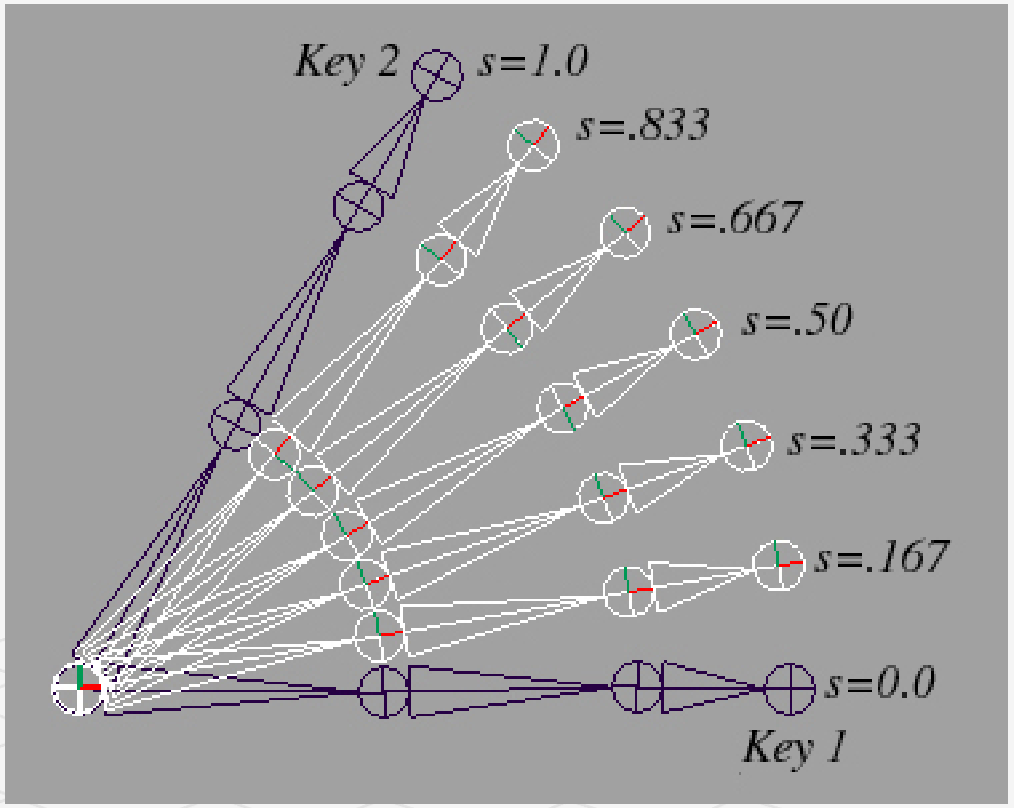
For times *t* between two key frames, say *Ki* and *Ki*+1, we interpolate between the two key frames *Ki* and *Ki*+1.

```
void BoneAnimation::Interpolate(float t,
XMFLOAT4X4& M)const
{
if( t <= Keyframes.front().TimePos )
{
XMVECTOR S =
XMLoadFloat3(&Keyframes.front().Scale);
XMVECTOR P =
XMLoadFloat3(&Keyframes.front().Translation);
XMVECTOR Q =
XMLoadFloat4(&Keyframes.front().RotationQuat)
;
XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
}
else if( t >= Keyframes.back().TimePos )
{
XMVECTOR S = XMLoadFloat3(&Keyframes.back().Scale);
XMVECTOR P = XMLoadFloat3(&Keyframes.back().Translation);
XMVECTOR Q = XMLoadFloat4(&Keyframes.back().RotationQuat);

XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));
}
else
{
for(UINT i = 0; i < Keyframes.size()-1; ++i)
{
if( t >= Keyframes[i].TimePos && t <= Keyframes[i+1].TimePos )
{
float lerpPercent = (t - Keyframes[i].TimePos) / (Keyframes[i+1].TimePos -
Keyframes[i].TimePos);

XMVECTOR s0 = XMLoadFloat3(&Keyframes[i].Scale);
XMVECTOR s1 = XMLoadFloat3(&Keyframes[i+1].Scale);

XMVECTOR p0 = XMLoadFloat3(&Keyframes[i].Translation);
XMVECTOR p1 = XMLoadFloat3(&Keyframes[i+1].Translation);

XMVECTOR q0 = XMLoadFloat4(&Keyframes[i].RotationQuat);
XMVECTOR q1 = XMLoadFloat4(&Keyframes[i+1].RotationQuat);

XMVECTOR S = XMVectorLerp(s0, s1, lerpPercent);
XMVECTOR P = XMVectorLerp(p0, p1, lerpPercent);
XMVECTOR Q = XMQuaternionSlerp(q0, q1, lerpPercent);

XMVECTOR zero = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
XMStoreFloat4x4(&M, XMMatrixAffineTransformation(S, zero, Q, P));

break;}}}}
```

Key frame interpolation

The key frames define the “key” poses of the animation. The interpolated values represent the values between the key frames.

Figure shows the in-between frames generated by interpolating from *Key 1* to *Key 2*.



XMMatrixAffineTransformation

After interpolation, we construct a transformation matrix because ultimately we use matrices for transformations in our shader programs. The

XMMatrixAffineTransformation function is declared as follows:

```
XMMATRIX
XMMatrixAffineTransformation(
XMVECTOR Scaling,
XMVECTOR RotationOrigin,
XMVECTOR RotationQuaternion,
XMVECTOR Translation);
```

Now that our simple animation system is in place, the next part of our demo is to define some key frames:

```
// Member data
float mAnimTimePos = 0.0f;
BoneAnimation mSkullAnimation;
```

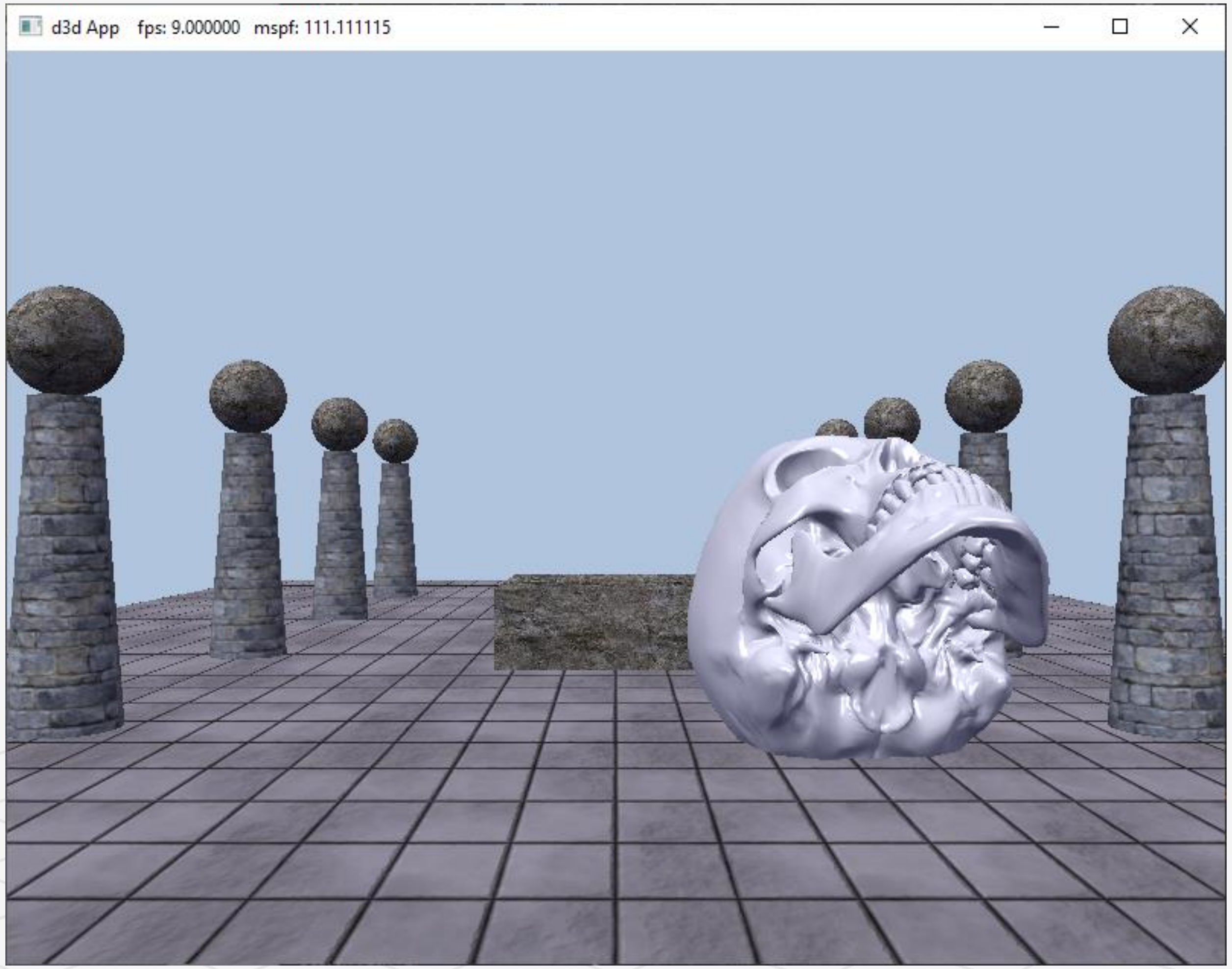
```
void QuatApp::DefineSkullAnimation()
{

    XMVECTOR q0 = XMQuaternionRotationAxis(XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(30.0f));
    XMVECTOR q1 = XMQuaternionRotationAxis(XMVectorSet(1.0f, 1.0f, 2.0f, 0.0f), XMConvertToRadians(45.0f));
    XMVECTOR q2 = XMQuaternionRotationAxis(XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f), XMConvertToRadians(-30.0f));
    XMVECTOR q3 = XMQuaternionRotationAxis(XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f), XMConvertToRadians(70.0f));

    mSkullAnimation.Keyframes.resize(5);
    mSkullAnimation.Keyframes[0].TimePos = 0.0f;
    mSkullAnimation.Keyframes[0].Translation = XMFLOAT3(-7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[0].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[0].RotationQuat, q0);

    mSkullAnimation.Keyframes[1].TimePos = 2.0f;
    mSkullAnimation.Keyframes[1].Translation = XMFLOAT3(0.0f, 2.0f, 10.0f);
    mSkullAnimation.Keyframes[1].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[1].RotationQuat, q1);

    mSkullAnimation.Keyframes[2].TimePos = 4.0f;
    mSkullAnimation.Keyframes[2].Translation = XMFLOAT3(7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[2].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[2].RotationQuat, q2);

    mSkullAnimation.Keyframes[3].TimePos = 6.0f;
    mSkullAnimation.Keyframes[3].Translation = XMFLOAT3(0.0f, 1.0f, -10.0f);
    mSkullAnimation.Keyframes[3].Scale = XMFLOAT3(0.5f, 0.5f, 0.5f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[3].RotationQuat, q3);

    mSkullAnimation.Keyframes[4].TimePos = 8.0f;
    mSkullAnimation.Keyframes[4].Translation = XMFLOAT3(-7.0f, 0.0f, 0.0f);
    mSkullAnimation.Keyframes[4].Scale = XMFLOAT3(0.25f, 0.25f, 0.25f);
    XMStoreFloat4(&mSkullAnimation.Keyframes[4].RotationQuat, q0);
}
```

Key frames

Our key frames position the skull at different locations in the scene, at different orientations, and at different scales.

Have fun experimenting with this demo by adding your own key frames or changing the key frame values.

Set all the rotations and scaling to identity, to see what the animation looks like when only position is animated.



QuatApp::Update

The last step to get the animation working is to perform the interpolation to get the new skull world matrix, which changes over time:

```
void QuatApp::Update(const GameTimer& gt)
{
    OnKeyboardInput(gt);

    mAnimTimePos += gt.DeltaTime();
    if(mAnimTimePos >= mSkullAnimation.GetEndTime())
    {
        // Loop animation back to beginning.
        mAnimTimePos = 0.0f;
    }

 mSkullAnimation.Interpolate(mAnimTimePos,mSkullWorld);

mSkullRitem->World = mSkullWorld;

mSkullRitem->NumFramesDirty = gNumFrameResources;
```

```
    // Cycle through the circular frame resource array.
    mCurrFrameResourceIndex = (mCurrFrameResourceIndex + 1) % gNumFrameResources;
    mCurrFrameResource = mFrameResources[mCurrFrameResourceIndex].get();

    // Has the GPU finished processing the commands of the current frame resource?
    // If not, wait until the GPU has completed commands up to this fence point.
    if(mCurrFrameResource->Fence != 0 && mFence->GetCompletedValue() <
mCurrFrameResource->Fence)
    {
        HANDLE eventHandle = CreateEventEx(nullptr, false, false, EVENT_ALL_ACCESS);
ThrowIfFailed(mFence->SetEventOnCompletion(mCurrFrameResource->Fence, eventHandle));
        WaitForSingleObject(eventHandle, INFINITE);
        CloseHandle(eventHandle);
    }

AnimateMaterials(gt);
UpdateObjectCBs(gt);
UpdateMaterialBuffer(gt);
UpdateMainPassCB(gt);
}
```