11 Deferred Rendering

11.1 Overview

As graphics hardware has become more and more generic and programmable, applications employing real-time 3D graphics have begun to explore alternatives to traditional rendering pipelines, in order to avoid their disadvantages. One of the most popular techniques currently in use is known as *deferred rendering*. This technique is primarily geared toward supporting large numbers of dynamic lights without a complex set of shader programs (see Figure 11.1). It has been successfully integrated into engines made by Crytek (Mittring, 2009), Naughty Dog (Balestra & Engstad, 2008) (Hable, 2010), and Guerrilla Games (Valient, 2007).

This chapter provides a brief introduction of deferred rendering that covers its advantages and disadvantages, while the following sections explore a basic Direct3D 11 implementation of both traditional deferred rendering and *light pre-pass deferred rendering*. The final three sections take things a step further and explore how Direct3D 11 features can be exploited to improve both the quality and performance of a deferred renderer.

11.1.1 Problems with Forward Rendering

One of the most important aspects in designing a renderer is determining how to handle lighting. Lighting is extremely important, since it deals with calculating the intensity and color of light reflecting off surface geometry at a particular point, which becomes the most

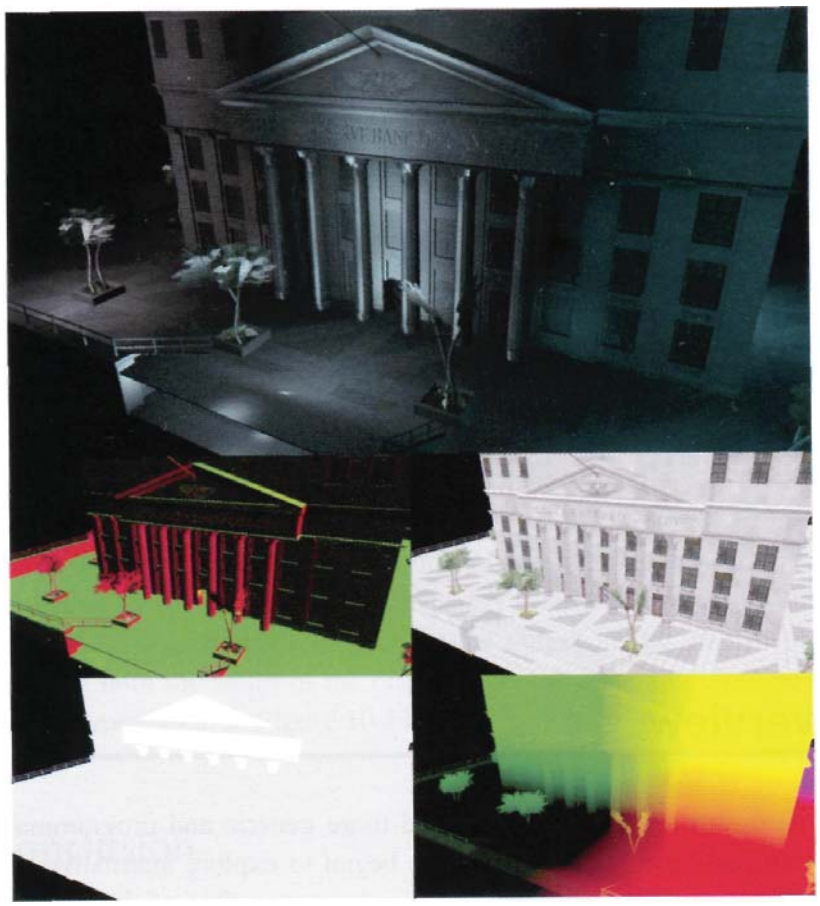

Figure 11.1. The final output and intermediate g-buffer textures of a deferred renderer. Model courtesy of Radioactive Software, LLC, www.radioactive-software.com. Created by Tomas Drinovsky, Danny Green.

significant factor in determining the final color of a pixel. Typically, it consists of the following steps:

1. Determine which lights need to be applied to a particular pixel, based on the light type and attenuation properties
2. Evaluate a *bidirectional reflection distribution function (BRDF)*[1] for each pixel, using material properties *(albedo),* surface properties *(normal* and *position),* and the properties of each light that affects the pixel.
3. Apply the results of a light visibility calculation, or in other words, determine whether or not a pixel is "in shadow."

[1] A *BRDF* is an equation used to evaluate the amount of light that reflects off a surface. Typically, a BRDF will use several input parameters, including the properties of the light, the material properties of the surface, and the position of the eye or camera. (Zink, Hoxley, Engel, Kornmann, & Suni)

The traditional way to handle this process is known as *forward rendering*. With forward rendering, geometry is rendered with various surface properties stored in the vertex attribute data, in textures, and in constant buffers. Then, for each pixel that is rasterized for the input geometry, the material data is used to apply the lighting equation for one or more lights. The results of evaluating the lighting equation are then output for each pixel, and possibly summed with the previous contents of the render target. This approach is straightforward and intuitive, and it was the overwhelmingly dominant approach for real-time 3D graphics before the advent of programmable graphics hardware. These fixed-function GPUs (and the earlier Direct3D APIs for using them) supported three different light types, as follows:

1. *Point lights*—these lights have an equal contribution in all directions and are attenuated based on the distance from the light source to the surface being lit. As a result, they have a spherical area of effect.

2. *Spot lights*—these have a direction associated with them, and the light contribution is attenuated based on the angle between the surface and the light direction. Consequently, spot lights have a conical area of effect.

3. *Directional lights*—unlike the other two light sources, this type is considered a "global" light source. This is because the position of the surface is not taken into account when determining the light contribution. In fact, directional light sources have no position, and are instead defined by only a direction. The light is treated as if it were a point light infinitely far away from the surface, in the direction specified for the light.

As real-time rendering engines have scaled to accommodate programmable hardware and more complex scenes, these basic light types have mostly remained, due to their flexibility and ubiquity. However, the means in which they are applied have begun to show to show their age. This is because with modern hardware and APIs, forward rendering exhibits several key weaknesses when it comes supporting large numbers of dynamic light sources.

The first disadvantage is that the application of light sources is tied to the granularity at which scene geometry is drawn. In other words, when we enable a light source, we must apply it to all the geometry that is rasterized during any particular draw call. At first it may not be obvious why this is a bad thing, especially for simple scenes with few lights. However, as the number of lights scales up, it becomes important to apply a light source selectively, to reduce the number of lighting calculations performed in the pixel shader. But since changing which lights are applied can only be done in between draw calls, we're limited in how much we can cull lights that aren't needed. So for cases where a light only affects a small portion of the geometry rasterized by a draw call, we must still apply that light to all geometry, since we can't selectively apply the light per-pixel. We could split the

meshes in the scene into smaller parts to improve granularity, but this increases the number of draw calls in a frame (and thus increases the CPU overhead). It would also increase the number of intersection tests required to determine if a light affects a mesh, which also increases the amount of work performed by the CPU. Another related consequence is that our ability to render multiple instances of geometry in a single batch is reduced, since the set of lights used has to be the same for all of the geometry instances in a batch. Since instances may appear in different areas of a scene, they typically will be affected by different combinations of lights.

The other major disadvantage of forward rendering is shader program complexity. Being able to handle various numbers of lights and light types with various material variations can require an explosion in the number of required shader permutations.[2] A high number of shader permutations is undesirable, since it increases memory usage, as well as overhead from switching between shader programs. Depending on how the permutations are compiled, they can also significantly increase compilation times. An alternative to using many shader programs is to use dynamic flow control, which has various GPU performance implications. Another alternative is to only render one light at a time and additively blend the result into the render target, which is known as *multipass rendering*. However, this approach requires paying the cost for transforming and rasterizing geometry multiple times. Even ignoring the issues associated with using many permutations, the resulting pixel shader programs can become very expensive and complex on their own. This is because of the need to evaluate material properties *and* perform necessary lighting and shadowing calculations for all active light sources. This makes the shader programs difficult to author, maintain, and optimize. Their performance is also tied to the overdraw of the scene, since overdraw results in shading pixels that aren't even visible. A *z*-only prepass can significantly reduce overdraw, but its efficiency is limited by the implementation of the Hi-*Z* unit in the hardware. Shader executions will also be wasted, since pixel shaders must run in 2×2 quads at a minimum, which can be particularly bad for highly-tessellated scenes with many small triangles.

These disadvantages collectively make it difficult to scale the number of dynamic lights in a scene while maintaining adequate performance for real-time applications. However if you look at the descriptions very carefully, you'll notice that all of these disadvantages stem from one primary problem that is inherent to forward rendering: lighting is tightly coupled with the rasterization of scene geometry. This means that if we were to decouple those two steps, we could potentially limit or completely bypass some of those disadvantages. But how can we do this? If we look at step 2 in the lighting process, we see that to calculate lighting, we need both the material properties and the geometric surface properties for our scene geometry. This means that if we have a step where we rendered out all

[2] The term *shader permutations* refers to compiling many variants of shader program using common HLSL source code as the base. Typically, conditional compilation and macros are used to enable or disable certain portions of the shader program, and the permutations are compiled with various combinations of those features

of these properties to a buffer (which we would call a *geometry buffer*, or *g-buffer* (Saito & Takahashi, 1990)), we could have a second step where we iterate through the lights in the scene and calculate the lighting values for each pixel. This is exactly the premise of deferred rendering, and we will explore many of the properties of this rendering technique throughout the remainder of this chapter.

11.1.2 The Deferred Rendering Pipeline

The first step of the deferred rendering pipeline (outlined in Figure 11.2) is to render all scene geometry into a g-buffen, which typically consists of several render target textures. The individual channels of these textures are used to store geometry surface information per-pixel, such as surface normal or material properties. The second step is the lighting phase. In this step, geometry representing the light's area of influence on the screen is rendered, and for each resulting fragment that is shaded, the g-buffer information is sampled from the render target textures. The g-buffer information is then combined with the light properties to determine the resulting lighting contribution to that pixel. This contribution is then summed with the contribution of all other light sources affecting that pixel, to determine the final, lit color of the surface.

Since deferred rendering avoids the primary drawback of forward rendering (lighting too tightly coupled with geometry), it has the following advantages:

The number of shader permutations is drastically reduced, since lighting and shadowing calculations can be moved into their own separate shader programs.

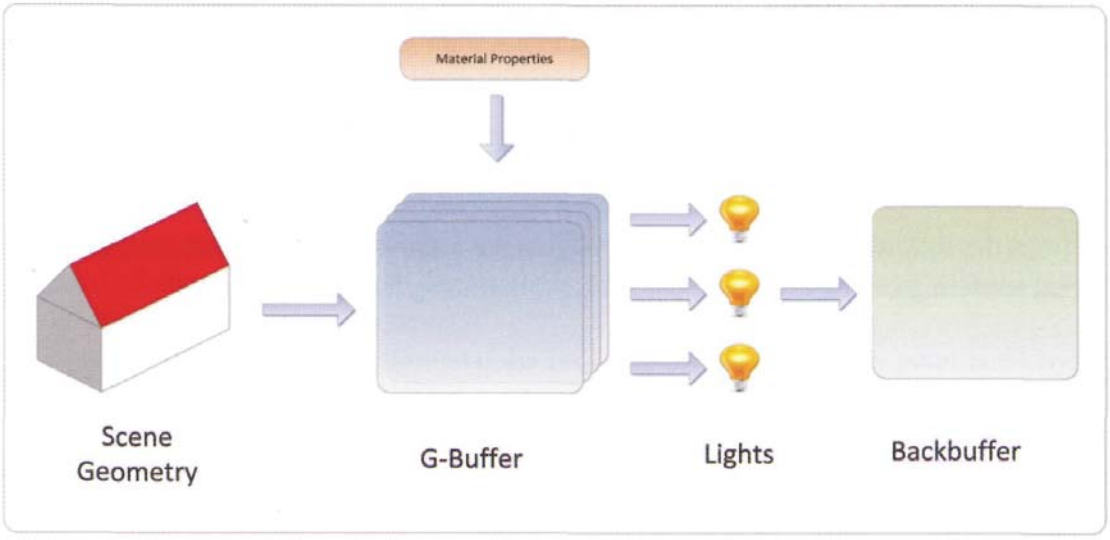

Figure 11.2. The deferred rendering pipeline.

Mesh instances can be batched more frequently, since lighting parameters are no longer needed when rendering scene geometry. Consequently, the active light sources do not need to be the same for all instances of a mesh.

Scene geometry only needs to be rendered once, since there's no need to resort to multipass rendering for lighting.

It's no longer necessary to perform CPU work to determine which lights affect different portions of the screen. Instead, bounding volumes or screen-space quads can be rasterized over the portion of the screen that a light affects. Depth and stencil testing can also be used to further reduce the number of shader executions.

The overall required shader and rendering framework architecture can be simplified, because lighting and geometry have been decoupled.

These advantages have caused deferred rendering to become extremely popular among modem real-time rendering engines. Unfortunately, the approach also comes with several drawbacks of its own:

- A significant amount of memory and bandwidth must be dedicated to the generation and sampling of the g-buffer, since it needs to store any information required to calculate lighting for that pixel.

  Using hardware MSAA is nontrivial, and generally requires that the lighting calculations be performed for each subsample. This can introduce a significantly increased computational burden.

- Transparent geometry can't be handled in the same manner as opaque geometry, since it can't be rendered into the g-buffer. This is because the g-buffer can only hold properties for a single surface, and rendering transparent geometry requires calculating the lighting for multiple overlapping per-pixel and combining the resulting color.

  Having different BRDFs for different materials is no longer straightforward, since the evaluation of a pixel's final color has been moved to the lighting pass.

In the following sections, we will talk about ways in which the advantages of deferred rendering can be exploited, while also minimizing it drawbacks.

11.2 Classic Deferred Rendering

In this section, we will work through an extremely basic deferred rendering implementation in D3D11. To maintain simplicity in this introduction to the topic, we will assume that

all materials use a *Blinn-Phong BRDF* (Zink, Hoxley, Engel, Kornmann, & Suni). We will also omit any performance optimizations for now, as these will be covered in the following section.

11.2.1 G-Buffer Layout

To start off, we must first design the layout and format of the g-buffer. As mentioned in the previous section, the g-buffer contains the scene geometry's surface properties that are needed to evaluate the BRDF. This means that we must examine the equations for our BRDF to determine which properties will be needed. For reference, the equations for the variant of Blinn-Phong that we will use are listed in Equation (11.1):

$$\begin{aligned} \text{DiffuseColor} &= \text{Albedo}_{\text{diffuse}} \times \text{LightColor}_{\text{diffuse}} \times (N \circ L), \\ \text{SpecularColor} &= \text{Albedo}_{\text{specular}} \times \text{LightColor}_{\text{specular}} \times (N \circ H)^{P}, \\ H &= \frac{L + V}{||L + V||}. \end{aligned} \tag{11.1}$$

In the above equations N is the surface normal direction vector, L is the direction vector from the surface to the light source, V is the direction vector from the surface to the camera, and P is the specular power (which is a measure of shininess/roughness in this lighting model). The *Albedo* terms refer to the reflectiveness of the material being shaded, and the specular and diffuse albedos are modulated with the specular and diffuse lighting contributions, respectively. To provide all of the parameters needed to complete the above equations, we will need to store diffuse albedo, specular albedo, the specular power, the surface normal vector, and the surface position vector in our g-buffer. To have our g-buffer contain all of the values required to evaluate the above equations, we will need to store 13 distinct floating-point values in our g-buffer render targets. Since we can store at most 4 values per texel in a texture,[3] we will need to use 4 render targets in our g-buffer. Figure 11.3 shows the formats of the render targets that we will use, along with the layout of the data within those render targets.

With these render target formats, the resulting footprint for each pixel is 64 bytes. This means that at 1280x720 resolution, the total size of the g-buffer is 56.25 megabytes, or 126.56 megabytes for 1920×1080. Keep in mind that this is *in addition* to the memory footprint of the back buffer and depth-stencil buffer, which is the minimum number of

[3] This is because the available texture element formats supply at most four components.

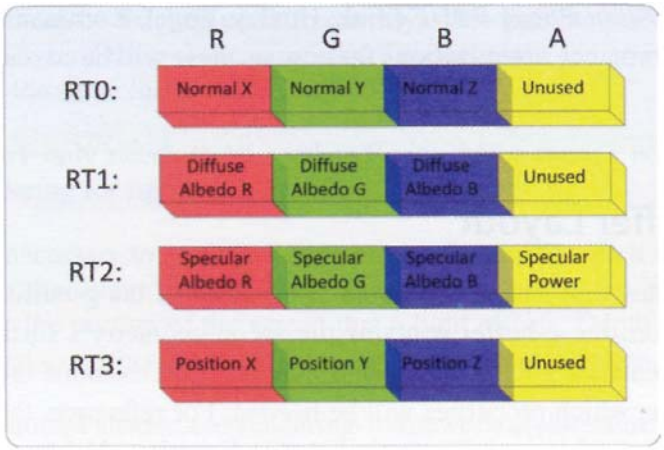

Figure 11.3. G-buffer layout.

resources required for a forward rendering setup. While this is obviously a very large footprint, various optimizations can be applied to greatly reduce the memory usage. See the section entitled "G-Buffer Attribute Packing" for more details.

11.2.2 Rendering the G-Buffer

G-Buffer Rendering without Normal Mapping

Direct3D 11 supports rendering to 8 simultaneous render targets, which means that we can fill our g-buffer by rendering a single pass for each mesh in our scene.[4] Since this pass only has to output material and surface properties, without calculating any lighting, we only require simple vertex and pixel shader programs. The vertex shader transforms vertex positions to clip space (for use by the rasterizer stage) and world space (for eventual storage in the g-buffer), and also transforms the vertex normal vectors to world space (also for eventual storage in the g-buffer). The pixel shader simply outputs the vertex position, vertex normal, diffuse albedo, specular albedo, and specular power. The code for these shader programs is provided in Listing 11.1.

[4] While a render target array could also hold the g-buffer data, it would require using the *SV_RenderTargetArrayIndex* system value semantic, and the geometry would be rasterized once for each of the render targets in the array. Since we aren't changing any of the viewing or perspective parameters for each render target, it doesn't make sense to perform multiple rasterizations. Thus, in this case, it is a better choice to use multiple render targets, instead of render target arrays.

```
// Constants
cbuffer Transforms
{
      matrix WorldMatrix;
      matrix WorldViewMatrix;
      matrix WorldViewProjMatrix;
};

cbuffer MatProperties
{
      float3  SpecularAlbedo;
      float  SpecularPower;
};
// Textures/Samplens,
Texture2D Diffus^Map          : register( t0 );
SamplerState AnisoSampler     : register( s0 );

// Input/Output structures
struct VSInput
{
      float4 Position           : POSITION;
      float2 TexCoord           : TEXCOORDS0;
      float3 Normal             : NORMAL;
};

struct VSOutput
{
      float4  PositionCS        : SV_Position;
      float2  TexCoord          : TEXCOORD;
      float3  NormalWS          : NORMALWS;
      float3  PositionWS        : POSITIONWS;
};

struct PSInput
{
      float4 PositionSS         : SV_Position;
      float2 TexCoord           : TEXCOORD;
      float3 NormalWS           : NORMALWS;
      float3 PositionWS         : POSITIONWS;
};

struct PSOutput
{
      float4 Normal             : SV_Target0;
      float4 DiffuseAlbedo      : SV_TargetI;
      float4 SpecularAlbedo     : SV_Target2;
      float4 Position           : SV_Target3;
};

// 6-Buffer vertex shader
VSOutput VSMain( in VSInput input )
{
      VSOutput output;
```

```
        // Convert position and normals to world space
        output.PositionWS = mul( input.Position, WorldMatrix ).xyz;
        output.NormalWS = normalize( mul( input.Normal, (float3x3)WorldMatrix ) );

        // Calculate the clip-space position
        output.PositionCS = mul( input.Position, WorldViewProjMatrix );

        // Pass along the texture coordinate
        output.TexCoord = input.TexCoord;

        return output;
}

// 6-Buffer pixel shader
PSOutput PSMain( in PSInput input )
{
        PSOutput output;

        // Sample the diffuse map
        float3 diffuseAlbedo = DiffuseMap.Sample( AnisoSampler, input.TexCoord ).rgb;

        // Normalize the normal after interpolation
        float3 normalWS = normalize( input.NormalWS );

        // Output our G-Buffer values
        output.Normal = float4( normalWS, 1.0f );
        output.DiffuseAlbedo = f1oat4( diffuseAlbedo, 1.0f );
        output.SpecularAlbedo = float4( SpecularAlbedo, SpecularPower );
        output.Position = float4( input.PositionWS, 1.0f );

        return output;
```

Listing 11.1. G-buffer generation shader code.

It should also be mentioned that the tessellation pipeline can also be used when rendering the g-buffer. G-buffer generation is the only pass where the scene geometry is being used, so if tessellation is desired, it should be used in this phase so that the g-buffer information reflects the finely tessellated meshes.

G-Buffer Rendering with Normal Mapping

Since the final surface normal vector is required for the lighting pass, any normal mapping[5] must be applied during the g-buffer pass. A common approach for forward rendering is to transform light positions and vectors to tangent space (Lengyel, 2001) in the vertex shader, and then apply lighting in that space to facilitate the use of normal maps. However, this

[5] *Normal mapping* is a technique that simulates more complex geometry by storing perturbed surface normal vectors in a texture, and then looks up these normal vectors in the pixel shader for use in lighting calculations.

approach doesn't work for deferred rendering, since the tangent frame isn't available in the lighting pass.[6] Instead, the normal map value must be transformed to view space or world space, so that it can be used with the light position and direction. In our implementation we perform the lighting calculations in world space, so we will transform the normal map value to world space, as well. The first step in doing this is to transform the vertex tangent frame to world space in the vertex shader. Then the per-vertex tangent frame is interpolated for each invocation of the pixel shader, which then samples the normal map value and uses the tangent frame to transform it to world space. The updated vertex and pixel shader programs are shown in Listing 11.2.

```
// Constants
cbuffer Transform
{
      matrix WorldMatrix;
      matrix WorldViewMatrix;
      matrix WorldViewProjMatrix;
};

cbuffer MatProperties
{
      float3 SpecularAlbedo;
      float SpecularPower;
};

// Textures/Samplers
Texture2D DiffuseMap            register( t0 );
Texture2D NormalMap             register( t1 );
SamplerState AnisoSampler       register( s0 );

// Input/Output structures
struct VSInput
{
      float4 Position     POSITION;
      float2 TexCoord     TEXCOORDS0;
      float3 Normal       NORMAL;
      float4 Tangent      TANGENT;
};

struct VSOutput
{
      float4 PositionCS           SV_Position;
      float2 TexCoord             TEXCOORD;
      float3 NormalWS             NORMALWS;
      float3 PositionWS           POSITIONWS;
      float3 TangentWS            TANGENTWS;
      float3 BitangentWS          BITANGENTWS;
};
```

[6] These tangent space basis vectors could be added to the g-buffer, but this would require as many as nine floating point values to describe the space correctly.

```
struct PSInput
{
    float4 PositionSS       : SV_Position;
    float2 TexCoord         : TEXCOORD;
    float3 NormalWS         : NORMALWS;
    float3 PositionWS       : POSITIONWS;
    float3 TangentWS        : TANGENTWS;
    float3 BitangentWS      : BITANGENTWS;
};

struct PSOutput
{
    float4 Normal           : SV_Target0;
    float4 DiffuseAlbedo    : SV_Target1;
    float4 SpecularAlbedo   : SV_Target2;
    float4 Position         : SV_Target3;
};

// G-Buffer vertex shader, with normal mapping
VSOutput VSMain( in VSInput input )
{
    VSOutput output;

    // Convert position and normals to world space
    output.PositionWS = mul( input.Position, WorldHatrix ).xyz;
    float3 normalWS = normalize( mul( input.Normal, (float3x3)WorldMatrix ) );
    output.NormalWS = normalWS;

    // Reconstruct the rest of the tangent frame
    float3 tangentWS = normalize( mul( input.Tangent.xyz,
                                       (float3x3)WorldMatrix ) );
    float3 bitangentWS = normalize( cross( normalWS, tangentWS ) )
                         * input.Tangent.w;

    output.TangentWS = tangentWS;
    output.BitangentWS = bitangentWS;

    // Calculate the clip-space position
    output.PositionCS = mul( input.Position, WorldViewProjMatrix );

    // Pass along the texture coordinate
    output.TexCoord = input.TexCoord;

    return output;
}

// G-Buffer pixel shader, with normal mapping
PSOutput PSMain( in PSInput input )
{
    PSOutput output;

    // Sample the diffuse map
    float3 diffuseAlbedo = DiffuseMap.Sample( AnisoSampler,
                                              input.TexCoord ).rgb;
```

```
    // Normalize the tangent frame after interpolation
    float3x3 tangentFrameWS = float3x3( normalize( input.TangentWS ),
                                        normalize( input.BitangentWS ),
                                        normalize( input.NormalWS ) )j

    // Sample the tangent-space normal map and decompress
    float3 normalTS = MormalMap.Sample( AnisoSampler, input.TexCoord ).rgb;
    normalTS = normalize( normalTS * 2.0f - 1.0f );

    // Convert to world space
    float3 normalWS = mul( normalTS, tangentFrameWS );

    // Output our G-Buffer values
    output.Normal = float4( normalWS, 1.0f );
    output.DiffuseAlbedo = float4( diffuseAlbedo, 1.0f );
    output.SpecularAlbedo = float4 ( SpecularAlbedo, SpecularPower );
    output.Position = float4( input.PositionWS, 1.0f );

    return output;
}
```

Listing 11.2. G-buffer generation shader code with normal mapping.

11.2.3 Rendering the Lights

After filling the g-buffer with the complete scene geometry, we perform a second rendering pass, where all lights in the scene are rendered. For each light, we will render a quad (comprised of two triangles) that covers the entire render target. The vertex shader used is extremely simple, as it merely passes the vertex position to the next stage. The pixel shader is more complex, and starts off by calculating a texture coordinate based on the screen-space pixel position (which is obtained using the SV_Position system value semantic). This texture coordinate is then used to sample the g-buffer textures, so that the material and surface parameters can be obtained. The shader then uses these parameters to evaluate the Blinn-Phong equations from Equation (11.1) to calculate the diffuse lighting contribution and the specular lighting contribution. These values are then summed, and an attenuation factor is applied, with the attenuation factor being based on the type of light source. For a point light, a simple linear distance attenuation model is used. For a spot light, the distance attenuation model is used, as well as an angular attenuation factor. For a directional light, no attenuation factor is applied, since directional lights are considered "global" light sources. Once the attenuation factor is applied, the resulting color value is output from the pixel shader with additive blending enabled. Listing 11.3 contains the shader code for performing these calculations

```
// Textures
Texture2D NormalTexture                 : register( t0 );
Texture2D DiffuseAlbedoTexture          : register( t1 );
Texture2D SpecularAlbedoTexture         : register( t2 );
Texture2D PositionTexture               : register( t3 );

// Constants
cbuffen LightParams
{
      float3  LightPos;
      float3  LightColor;
      float3  LightDirection;
      float2  SpotlightAngles;
      float4  LightRange;
}i

cbuffer CameraParams
{
      float3   CameraPos;
};

// Helper function for extracting G-Buffer attributes
void GetGBufferAttributes( in float2 screenPos, out float3 normal,
                           out float3 position,
                           out float3 diffuseAlbedo, out float3 specularAlbedo,
                           out float specularPower )
{
      // Determine our indices for sampling the texture based on the current
      // screen position
      int3 sampleIndices = int3( screenPos.xy, 0 );

      normal = NormalTexture.Load( sampleIndices ).xyz;
      position = PositionTexture.Load( sampleIndices ).xyz;
      diffuseAlbedo = DiffuseAlbedoTexture.Load( sampleIndices ).xyz;
      float4 spec = SpecularAlbedoTexture. Load( sampleIndices );

      specularAlbedo = spec.xyz;
      specularPower = spec.w;
}

// Calculates the lighting term for a single G-Buffer texel
float3 Calctighting( in float3 normal,
                     in float3 position,
                     in float3 diffuseAlbedo,
                     in float3 specularAlbedo,
                     in float specularPower )
{
      // Calculate the diffuse term
      float3 L = 0;
      float attenuation = 1.0f;
      #if POINTLIGHT || SPOTLIGHT
            // Base the the light vector on the light position
            L = LightPos - position;
```

```
            // Calculate attenuation based on distance from the light source
            float dist = length( L );
            attenuation = max( 0, 1.0f - ( dist / LightRange.x ) );

            L /= dist;
        #elif DIRECTIONALLIGHT
            // Light direction is explicit for directional lights
            L = -LightDirection;
        #endif

        #if SPOTLIGHT
            // Also add in the spotlight attenuation factor
            float3 L2 = LightDirection;
            float rho = dot( -L, L2 );
            attenuation *= saturate( ( rho - SpotlightAngles.y )
                                     / ( SpotlightAngles.x -
                                         SpotlightAngles.y ) );
    #endif
        float nDotL = saturate( dot( normal, L ) );
        float3 diffuse = nDotL * LightColor * diffuseAlbedo;

        // Calculate the specular term
        float3 V = CameraPos - position;
        float3 H = normalize( L + V );
        float3 specular = pow( saturate( dot( normal, H ) ), specular-Power )
                                    * LightColor * specularAlbedo.xyz * nDotL;

        // Final value is the sum of the albedo and diffuse with attenuation applied
        return ( diffuse + specular ) * attenuation;
}

// Lighting pixel shader
float4 PSMain( in float4 screenPos : SV_Position ) : SV_Target0
{
        float3 normal;
        float3 position;
        float3 diffuseAlbedo;
        float3 specularAlbedo;
        float specularPower;

        // Sample the G-Buffer properties from the textures
        GetGBufferAttributes( screenPos.xy, normal, position, diffuseAlbedo,
                              specularAlbedo, specularPower );

        fioat3 lighting = CalcLighting( normal, position, diffuseAlbedo,
                                        specularAlbedo, specularPower );

        return float4( lighting, 1.0f );
}
```

Listing 11.3. Lighting shader code.

Since the blend state used has additive blending enabled, the output is summed with the previous contents of the render target. Thus, the output from each light is summed to create the final color value for each pixel.

11.3 Light Pre-Pass Deferred Rendering

Deferred rendering has several advantages, but it also suffers in performance and flexibility due to the need to have all of the surface and material properties contained in the g-buffer. Storing many attributes in the g-buffer consumes a great deal of memory and bandwidth (particularly when MSAA is used), and also makes it difficult to implement different types of materials and lighting models, light pre-pass deferred rendering(Engel, 2009) is a technique that attempts to alleviate these problems somewhat by further splitting the lighting pipeline into three steps, as opposed to two. The end result is that the g-buffer can be made much smaller, and the lighting pass is less expensive, but all deferred scene geometry must be rendered twice, instead of once. The basic steps are as follows:

1. Render a minimal g-buffer with only the surface properties required to evaluate the core portions of the lighting equation.
2. Evaluate the core portion of the lighting equation for all lights in the scene by rendering geometry covering the affected pixels, and sum the results into a light buffer.
3. Render the scene geometry a second time, in which the light buffer value is sampled and the material albedos are applied to determine the final surface color.

This section will walk through a basic implementation of a light pre-pass deferred renderer without optimizations, similar to the outline in the previous section. It should be noted that light pre-pass deferred rendering is often referred to as *deferred lighting* to distinguish it from *deferred rendering* or *deferred shading.* This emphasizes the fact that the basic lighting calculations are done in a deferred step, but the final shading step is done while rasterizing the scene geometry.

11.3.1 Light Pre-Pass G-Buffer Layout

As mentioned in the overview, with light pre-pass deferred rendering, we can render a minimal g-buffer. Essentially, we only want the surface properties required to evaluate the core diffuse and specular components of the Blinn-Phong BRDF, without the albedos.

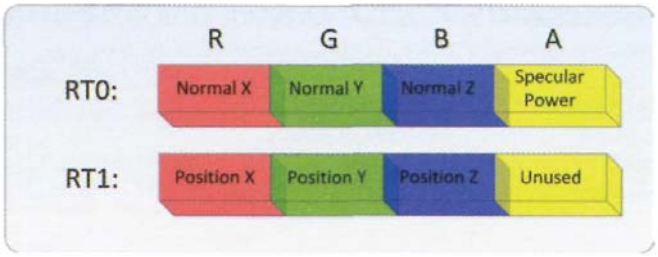

Figure 11.4. G-buffer layout for light prepass deferred rendering.

Thus, we only need surface position, surface normal vector, and specular power in our g-buffer,Figure 11.4 shows the resulting layout in our render targets:

I For our basic implementation, this leaves us with only 2 render targets, instead of the 4 requrred for the classic deferred Tenderer. For our unoptimized implementation, it also cuts our g-buffer memory footprint in half, leaving it at 32 bytes instead of 64. It should be noted that it is possible to have only 1 render target if position is reconstructed from the depth buffer instead of being explicitly stored in the g-buffer (see the section entitled "G-Buffer Attribute Packing" for details). This can provide an advantage on hardware on which using multiple render targets has a significant performance penalty.

11.3.2 Rendering the G-Buffer

The approach for rendering the g-buffer is very similar to the one used for the classic deferred rendering implementation. The only difference is that we no longer need to write the diffuse albedo or the specular albedo. The shader program code for doing this is listed in Listing 11.4.

```
// Textures
Texture2D     NormalMap        : register( t0 );
Sampler-State  AnisoSampler    : register( s0 );

// Constants
cbuffer Transforms
{
      matrix WorldMatrix;
      matrix WorldViewMatrix;
      matrix WorldViewProjMatrix;
};

cbuffer MaterialProperties
{
      float  SpecularPower;
};
```

```
// Input/output structures
struct VSInput
{
      float4 Position          : POSITION;
      float2 TexCoord          : TEXCOORDS0;
      float3 Normal            : NORMAL;
      float4 Tangent           : TANGENT;
};

struct VSOutput
{
      float4 PositionCS        : SV_Position;
      float2 TexCoord          : TEXCOORD;
      float3 NormalWS          : NORMALWS;
      float3 TangentWS         : TANGENTWS;
      float3 BitangentWS       : BITANGENTWS;
      float3 PositionWS        : POSITIONWS;
};

struct PSInput
{
      float4 PositionSS        : SV_Position;
      float2 TexCoord          : TEXCOORD;
      float3 NormalWS          : NORMALWS;
      float3 TangentWS         : TANGENTWS;
      float3 BitangentWS       : BITANGENTWS;
      float3 PositionWS        : POSITIONWS;
};

struct PSOutput
{
      float4 Normal            : SV_Target0;
      float4 Position          : SV_Target1;
};

// G-Buffer vertex shader for light prepass deferred rendering
VSOutput VSMain( in VSInput input )
{
      VSOutput output;

      // Convert normals to world space
      float3 normalWS = normalize( mul( input.Normal, (float3x3)WorldMatrix ) );
      output.NormalWS = normalWS;

      // Reconstruct the rest of the tangent frame
      float3 tangentWS = normalize( mul( input.Tangent.xyz,
                                         (float3x3)WorldMatrix ) );
      float3 bitangentWS = normalize( cross( normalWS, tangentWS ) )
                                         * input.Tangent.w;

      output.TangentWS = tangentWS;
      output.BitangentWS = bitangentWS;

      // Calculate the world-space position
```

```
        output.PositionWS = mul( input.Position, WorldMatrix ).xyz;

        // Calculate the clip-space position
        output.PositionCS = mul( input.Position, WorldViewProjMatrix );

        // Pass along the texture coordinate
        output.TexCoord = input.TexCoord;

        return output;
}

// G-Buffer pixel shader for light prepass deferred rendering
PSOutput PSMain( in PSInput input )
{
        //.Normalize the tangent frame after interpolation
        float3x3 tangentFrameWS = float3x3( normalize( input.TangentWS ),
                                            normalize( input.BitangentWS ),
                                            normalize( input.NormalWS ) );

        // Sample the tangent-space normal map and decompress
        float3 normalTS = NormalMap.Sample( AnisoSampler, input.TexCoord ).rgb;
        normalTS = normalize( normalTS * 2.0f - 1.0f );

        // Convert to world space
        float3 normalWS = mul( normalTS, tangentFrameWS );

        // Output our G-Buffer values
        PSOutput output;
        output.Normal = float4( normalWS, SpecularPower );
        output.Position = float4( input.PositionWS, 1.0f );
        return output;
}
```

Listing 11.4. G-buffer generation shader code for light prepass deferred rendering.

11.3.3 Rendering the Light Buffer

For the lighting pass, we will only evaluate a portion of the lighting equation, and will the result to the render target. The simplified equations that we will use are listed below in Equation (11.2):

$$\begin{aligned} \text{DiffuseLight} &= \text{LightColor}_{\text{diffuse}} \times (N \circ L), \\ \text{SpecularLight} &= \text{LightColor}_{\text{specular}} \times (N \circ H)^{p}, \\ H &= \frac{L+V}{\|L+V\|}. \end{aligned} \tag{11.2}$$

Since *DiffuseLight* and *SpecularLight* each have 3 components, storing the full values in our light buffer would require storing 6 individual floating-point values. This would require writing to at least two render targets simultaneously, which is often not optimal from a performance and memory-consumption point of view. Thus, a common optimization is to store only the specular intensity rather than the color, which allows the light buffer to consist of a single 4-component texture. This approach is taken in the lighting shader code in Listing 11.5.

```
// Textures
Texture2D NormalTexture        : register( t0 );
Texture2D PositionTexture      : register( t1 );

// Constants
cbuffer LightParams
{
      float3 LightPos;
      float3 LightColor;
      float3 LightDirection;
      float2 SpotlightAngles;
      float4 LightRange;
};

cbuffer CameraParams
{
      float3 CameraPos;
}

// Vertex shader for lighting pass of light prepass deferred rendering
float4 VSMain( in float4 Position : POSITION ) : SV_Position
{
      // Just pass along the position of the full-screen quad
      return Position;
}

// Helper function for extracting G-Buffer attributes
void GetGBufferAttributes( in float2 screenPos, out float3 normal,
                           out float3 position, out float specularPower )
{
      // Determine our indices for sampling the texture based on the current
      // screen position
      int3 sampleIndices = int3( screenPos.xy, 0 );

      float4 normalTex = NormalTexture.Load( sampleIndices );
      normal = normalTex.xyz;
      specularPower = normalTex.w;
      position = PositionTexture.Load( sampleIndices ).xyz;
}

// Calculates the lighting term for a single G-Buffer texel
float4 CalcLighting( in float3 normal, in float3 position, in float specularPower )
```

```
{
        // Calculate the diffuse term
        float3 L = 0;
        float attenuation = 1.0f;
        #if POINTLIGHT || SPOTLIGHT
                // Base the the light vector on the light position
                L = LightPos - position;

                // Calculate attenuation based on distance from the light source
                float dist = length ( L );
                attenuation = max( 0, 1.ef - ( dist / LightRange.x ) );

                L /= dist;
        #elif DIRECTIONALLIGHT
                // Light direction is explicit for directional lights
                L = -LightDirection;
        #endif

       #if  SPOTLIGHT
                // Also add in the spotlight attenuation factor
                float3 L2 = LightDirection;
                float rho = dot( -L, L2 );
                attenuation *= saturate( ( rho - SpotlightAngles.y )
                                / ( SpotlightAngles.x - SpotlightAngles.y ) );
        #endif

        float nDotL = saturate( dot( normal, L ) );
        float3 diffuse = nDotL * LightColor * attenuation;

        // Calculate the specular term
        float3 V = CameraPos - position;
        float3 H = normalize( L + V );
        float specular = pow( saturate( dot( normal, H ) ), specularPower )
                                                * attenuation * nDotL;

        // Final value is diffuse RGB + mono specular
        return float4( diffuse, specular );
}

// Pixel shader for lighting pass of light prepass deferred rendering
float4 PSMain( in float4 screenPos : SV_Position ) : SV_Target0
{
        float3 normal;
        float3 position;
        float  specularPower;

        // Get the G-Buffer values
        GetGBufferAttributes( screenPos.xy, normal, position, specularPower );

        return CalcLighting( normal, position, specularPower );
}
```

Listing 11.5. Light buffer generation pixel shader code.

Once the light buffer is rendered, it is possible to apply screen-space operations to the contents of the light buffer to simulate more complex light interactions. For example, a bilateral blur can be used to simulate subsurface scattering for skin and other translucent materials (Mikkelsen, 2010).

11.3.4 Rendering the Final Pass

In the final pass, opaque scene geometry is rendered for a second time. The vertex shader is quite simple, as it only needs to pass along the mesh's diffuse map texture coordinate to the pixel shader. The pixel shader samples the light buffer based on the screen-space position of the pixel being shaded, and the values in the light buffer are combined with material albedos to determine the final color of the surface. The shader in Listing 11.6 demonstrates this concept.

```
// Textures
Texture2D     DiffuseMap       : register( t0 );
Texture2D     LightTexture     : register( t1 );
SamplerState  AnisoSampler     : register( s0 );

// Constants
cbuffer Transforms
{
      matrix WorldMatrix;
      matrix WorldViewMatrix;
      matrix WorldViewProjMatrix;
};
cbuffer MaterialProperties
{
      float3 SpecularAlbedoj
}

// Input/Output structures
struct VSInput
{
      float4 Position   : POSITION;
      float2 TexCoord   : TEXCOORDS0;
};

struct VSOutput
{
      float4 PositionCS : SV_Position;
      float2 TexCoord   : TEXCOORD;
};

struct PSInput
```

```
{
        ftoat4  ScreenPos   : SV_Position;
        float2  TexCoord    : TEXCOORD;
};

// Vertex shader for final pass of light prepass deferred rendering
VSOutput VSMain( in VSInput input )
{
        VSOutput output;

        // Calculate the clip-space position
        output.PositionCS = mul( input.Position, WorldViewProjMatrix );

        // Pass along the texture coordinate
        output.TexCoord = input.TexCoord;

        return output;
}

// Pixel shader for final pass of light prepass deferred rendering
float4 PSHain( in PSInput input ) : SV_Target0
{
        // Sample the diffuse map
        float3 diffuseAlbedo = DiffuseMap.Sample( AnisoSampler, input.TexCoord ).rgb;

        // Determine our indices for sampling the texture based on the current
        // screen position
        int3 sampleIndices = int3( input.ScreenPos.xy, 0 );

        // Sample the light target
        float4 lighting = LightTexture.Load( sampleIndices );

        // Apply the diffuse and specular albedo to the lighting value
        float3 diffuse = lighting.xyz * diffuseAlbedo;
        float3 specular = lighting.w * SpecularAlbedo;

        // Final output is the sum of diffuse + specular
        return float4( diffuse + specular, 1.0f );
}
```

Listing 11.6. Shader code for final pass.

It should be noted that other terms that contribute to the final pixel color can also be added in, such as emissive lighting (glow), Fresnel/rim lighting, or reflection maps. This makes light pre-pass deferred rendering more natural for implementing these sorts of material variations.

11.4 Optimizations

The deferred rendering implementations that we have seen so far would present suboptimal performance characteristics in many situations, since their aim was to demonstrate the core concepts of deferred rendering. To make the approach practical for use in real-time applications, it is often necessary to leverage the flexibility and programmability of the Direct3D 11 pipeline to improve the performance. The following sections will focus on various methods for reducing the required memory and bandwidth usage associated with rendering and sampling the g-buffer, as well as minimizing the number of pixel shader executions that are needed in the lighting pass.

11.4.1 G-Buffer Attribute Packing

One of the primary disadvantages of using a deferred approach to rendering is that surface and material properties must be rendered out to one or more render targets. A significant amount of bandwidth must be dedicated to writing the values during the g-buffer generation pass, as well as when sampling the textures during the lighting pass. As a result, reducing the memory footprint of the g-buffer is a simple and effective method for optimizing the performance of a deferred renderer. This is generally true even when some extra math is performed in the shader programs to pack or unpack the values from a compressed format, since almost all modern GPUs have a large number of ALUs relative to their bandwidth and texturing units. With this in mind, we will look at the various types of data typically stored in a g-buffer and determine a compressed format for storage, based on the typical range of values, the precision required to avoid unacceptable artifacts, and the expense of packing and unpacking the data.

Normal Vectors

For compressing normal vectors, we can take advantage of the fact that a normal vector is a unit-length direction vector. This greatly restricts the domain of the normal values, which can allow us to use a signed 8-bit integer format (DXGI_FORMAT_R8G8B8A8_SNORM) for storage, instead of a floating point format. It can also allow us to store the direction in a representation that requires only 2 values instead of 3, thus dropping a component entirely.

Perhaps the most obvious approach is to convert the normal vector to Spherical Coordinates and omit p, since it is implicit that $p = 1$ for a normalized direction vector. The equations for converting a vector from a Cartesian coordinate system to a spherical coordinate system are well-known and are trivial to implement in HLSL. The conversion can also be algebraically simplified for the case where $p = 1$, reducing the number

of shader instructions required for compression and decompression (Wilson, 2008). This simplification is demonstrated by the shader code in Listing 11.7.

```
float2 CartesianToSpherical(float3 cartesian)
{
    float2 spherical;

    spherical.x = atan2(cartesian.y, cartesian.x) / 3.14159f;
    spherical.y = cartesian.z;

    return spherical;
}

float3 SpherlcalToCartesian(float2 spherical)
{
    float2 sinCosTheta, sinCosPhi;

    sincos(spherical.x * 3.14159f, sinCosTheta.x, sinCosTheta.y);
    sinCosPhi = float2(sqrt(1.0 - spherical.y * spherical.y)., spherical.y);

    return float3(sinCosTheta.y * sinCosPhi.x,
                  sinCosTheta.x * sinCosPhi.x,
                  sinCosPhi.y);
}
```

Listing 11.7. Optimized spherical coordinate conversion for normals.

The main downside of using spherical coordinates is that they require use of the trigonometric intrinsics *sin(), cos(),* and *atan2().* These operations are among a special group of arithmetic operations known as *transcendentals*, and they often use hardware resources that are somewhat scarce on modern GPUs. As a result, it can often be desirable from a performance point of view to avoid their usage altogether when an alternative method exists.

Another approach is to simply store the *X*- and *Y*- components of the Cartesian normal vector and drop the Z component (Valient, 2007), since the Z component of a unit vector can be reconstructed as long as *X*, *Y*, and the sign of Z are known. Equation (11.3) shows the equation used for this reconstruction:

$$z = \pm\sqrt{1 - (x^2 + y^2)}. \tag{11.3}$$

It is possible to pack the sign bit into the stored *X* or *Y* values, making this technique suitable for normal vectors in any coordinate space. However some implementations have stored view space normal vectors, and then assumed that the sign of Z is always negative (or positive, if a right-handed coordinate system is used). At first this seems like a plausible assumption to make, since an object must be facing the viewer if it is visible. However, it

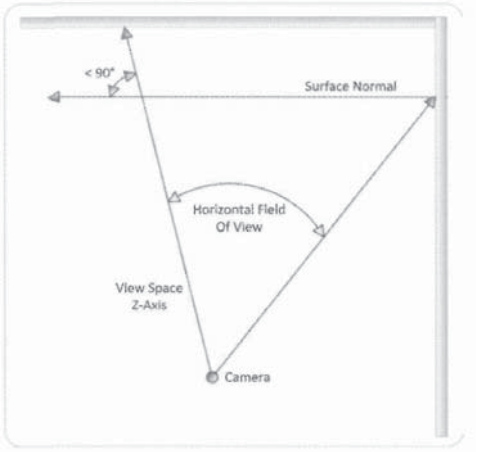

Figure 11.5. Negative view-space normals.

is important to note while it is *generally* true that the sign of Z will be negative for a view-space normal, with a perspective projection it is possible for surfaces to be rasterized that face in the positive Zdirection (Lee, 2009). This is illustrated in Figure 11.5.

In addition, normal maps completely override the vertex normal, making it possible for the normal to face in any direction. This will cause visual artifacts due to incorrect lighting results, which may or may not be unacceptable, depending on the error introduced by having an incorrect sign.

A third approach to normal vector compression is to use a *sphere map transformation* (Mittring, 2009).[7] This transform was originally developed for mapping a reflection vector to the [0, 1] range, but it also works for a normal vector. It essentially works by storing a 2D location on a map, where each location corresponds to the normal vector on a sphere. The shader code for performing the packing and unpacking is shown in Listing 11.8.

```
float2 SpheremapEncode( float3 normal )
{
    return normalize( normal.xy ) * ( sqrt( -normal.z * 0.5f + 0.5f ));
}

float3 SpheremapDecode( float2 encoded )
{
    float4 nn = float4( encoded, 1, -1 );
    float 1 = dot( nn.xyz, -nn.xyw );
    nn.z = 1;
    nn.xy *= sqrt( 1 );
    return nn.xyz * 2 + float3( 0, 0, -1 );
}
```

Listing 11.8. Sphere map transformation shader code.

[7] A discussion of sphere mapping and its properties are available in (Zink, Hoxley, Engel, Kornmann, & Suni).

Diffuse Albedo

Diffuse albedo is a simple case, since the source data is typically a color value of the range [0,1]. This means that we can use an unsigned, normalized, 8-bit integer format such as DXGI_FORMAT_R8G8B8A8_UNORM. It may also be desirable to store the values in sRGB color space,[8] since this is typically the storage format for diffuse textures. This requires using a format such as DXGI_FORMAT_R8G8B8A8_UNORM_SRGB, which causes the hardware to perform the sRGB conversion on the value output from the pixel shader. Alternatively, a 10-bit format such as DXGI_FORMAT_R10G10B16A2_UNORM can provide additional precision if the fourth component isn't needed for other data.

Specujar Albedo and Power

Specular albedo is a similar case to diffuse albedo, in that it doesn't require a great deal of precision. The 255 discrete values provided by an 8-bit integer are generally adequate for both the albedo and the power. Additionally, it is common to store only a monochrome specular value, rather than RGB components. This allows us to store only 2 values for specular albedo, as opposed to 4.

Position

Position is a value that typically requires high precision, since it can have a potentially large domain and is used for high-frequency shadow computations. For storing the *XYZ* components of a world or view space position, even a 16-bit floating point format is generally inadequate for avoiding artifacts. Fortunately, there is an alternative to storing the full *XYZ* value. When executing a pixel shader, the screen space *XY* position is implicit and available to shader through the SV_Position semantic. By using the view and projection matrices used when rendering the scene geometry, it is possible to reconstruct the view space or world space position of a pixel from only a single value.

To begin with, we will discuss the basics of a perspective projection. For any pixel on the screen, there is an associated direction vector that begins at the camera position and points towards the pixel's position on the far clipping plane. You can calculate this direction vector by using the screen-space *XY* position of the pixel to linearly interpolate between the positions of the corners on the view frustum's far clipping plane, subtracting the camera position, and then normalizing (you don't have to subtract the camera position if you're doing this in view space, since the camera position is located at the origin of view space). If geometry is rasterized at that pixel position, this means that the surface at that pixel lies somewhere along the vector from the origin to that pixel. The distance along that

[8] The *sRGB* color space is a standard RGB color space often used by image files and display devices. Using sRGB causes more precision to be used for darker color values, which matches the human eye's natural sensitivity to those color regions.

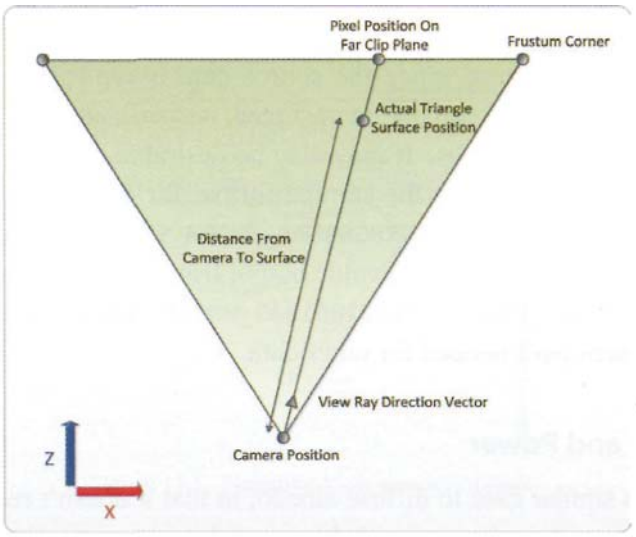

Figure 11.6. Reconstructing surface position using view ray and camera distance.

vector will vary, depending on how far that geometry is from the camera, but the direction is always the same. This is very similar to the method used for casting primary rays in a ray tracer: for a pixel on the near or far clip plane, get the direction from the camera to that pixel, and check for intersections. What this ultimately means is that if we have the screen space pixel position and the camera position, we can figure out the position of the triangle surface if we have the distance from the camera to the surface. Refer to Figure 11.6 for an illustration of this concept.

We're now ready to work out a simple implementation of this concept in a deferred Tenderer. The first step will be in our g-buffer pass, where we'll store the distance from the camera to the triangle surface in a single floating-point value. The simplest and cheapest way to do this is to transform the vertex position to view space in the vertex shader (since doing this implicitly makes the position relative to the camera position, as described above) and then compute the magnitude of the resulting position vector in the pixel shader. Listing 11.9 provides the g-buffer shader code for calculating camera distance.

```
// -- G-Buffer vertex shader
// Calculate view space position of the vertex and pass it to the pixel
// shader
output.PositionVS = mul(input.PositionOSj WorldViewMatrix).xyz;

// -- 6-Buffer pixel shader --
// Calculate the length of the view space position to get the distance from
// camera->surface
output.Distance.x = length(input.PositionVS);
```

Listing 11.9. G-buffer shader code for camera distance calculation.

The next step is to calculate the view ray vector during our lighting pass. We'll do this by first determining the vector from the camera to the vertex position in the vertex shader. Then in the pixel shader, we'll normalize this vector to get the final view ray vector. This vector is then multiplied with the distance value sampled from the g-buffer and added to the camera position to reconstruct the original surface position. Note that the vertex position, camera position, and resulting camera ray can be in any coordinate space, as long as the same space is used for all three of them. This allows reconstructing either a view space or a world space position. The code in Listing 11.10 uses world space for simplicity of presentation.

```
// -- Light vertex shader --
#if,VOLUMES
      // Calculate the world space position for a light volume
      float3 positionWS = mul(input.PositionOS, WorldMatrix);
#elif QUADS
      // Calculate the world space position for a full-screen quad (assume input
      // vertex coordinates are in [-1,1] post-projection space)
      float3 positionWS = mul(input.PositionOS, InvViewProjMatrix);
#endif

// Calculate the view ray
output.ViewRay = positionWS - CameraPositionWS;

// -- Light Pixel shader --
// Normalize the view ray, and apply the distance to reconstruct position
float3 viewRay = normalize(input.ViewRay);
float viewDistance = DistanceTexture.Sample(PointSampler, texCoord);
float3 positionWS = CameraPositionWS + viewRay * viewDistance;
```

Listing 11.10. Light shader code for view ray calculation and position reconstruction.

This provides us with a flexible and fairly efficient method for reconstructing position from only a single high-precision value stored in the g-buffer. However, we can still make further optimizations if we restrict ourselves to view space, since in view space the view frustum is aligned with the Z-axis.

When reconstructing a view space position, we can extrapolate the view ray until it intersects with the far clipping plane. Computing this intersection point is trivial, since it's the point where the Z component is equal to the far clip distance. When rasterizing a quad for a directional light, it's even simpler, since the frustum corner position can be linearly interpolated. With the Z component of the view ray set to a known value, it is no longer necessary to normalize the view ray vector in the pixel shader. Instead, we can multiply it by a value that scales along the camera's z-axis to get the final reconstructed position. In the case where Z is the far clip distance, we want to scale by a ratio of the original surface depth relative to the far clip plane. In other words, the surface's view space Z divided by the far

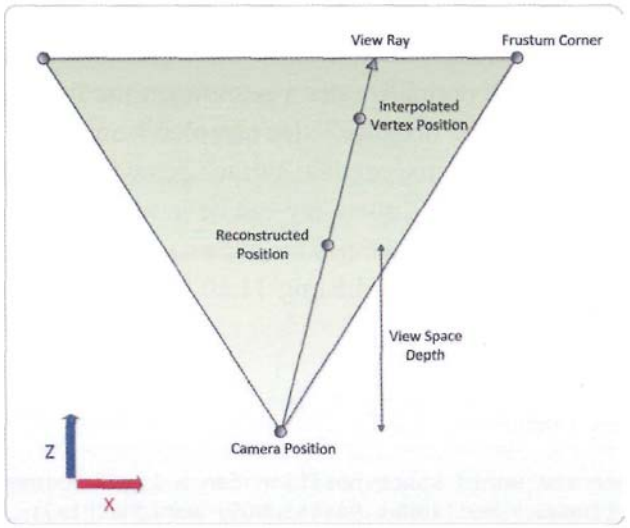

Figure 11.7. Optimized view-space position reconstruction.

clip distance. This scaling value can be calculated in the g-buffer pass and stored in the g-buffer, so that it can be used in the lighting pass. This technique is illustrated in Figure 11.7.

```
// -- G-Buffer vertex shader --
// Calculate view space position of the vertex and pass it to the pixel
shader
output.PositionVS = mul(input.PositionOS, WorldViewMatrix).xyz;

// -- G-Buffer pixel shader --
// Divide view space Z by the far clip distance
output.Depth.x = input.PositionVS.z / FarClipDistance;

// -- Light vertex shader --
#if VOLUMES
      // Calculate the view space vertex position
      output.PositionVS = mul(input.PositionOS, WorldViewMatrix);
#elif QUADS
      // Calculate the view space vertex position (you can also just directly
      // map the vertex to a frustum corner to avoid the transform)
      output.PositionVS = mul(input.PositionOS, InvProjMatrix);
#endif

// Light Pixel shader
#if VOLUMES
      // Extrapolate the view space position to the far clip plane
      float3 viewRay = float3(input.PositionVS.xy *
                                    (FarClipDistance / input.PositionVS.z),
                        FarClipDistance);
#elif QUADS
```

```
// For a directional lightj the vertices were already on the far clip plane so
// we don't need to extrapolate
      float3 viewRay = input.PositionVS.xyz;
#endif

// Sample the depth and scale the view ray to reconstruct view space position
float normalizedDepth = DepthTexture.Sample(PointSampler, texCoord).xj
float3 positionVS = viewRay * normalizedDepth;
```

Listing 11.11. Shader code for optimized view space position reconstruction.

Avoiding normalization of the view ray in the pixel shader reduces the number of math operations, particularly in the directional light case where the view ray is already extrapolated. Another point to be aware of is that the depth value stored in the g-buffer is always ofjthe range [0, 1]. This means you can store it in a normalized integer format (such as DXGI_F0RMAT_R16_UNORM) without having to do any rescaling after sampling it in the pixel shader.

While the above techniques work well when a depth or distance value is explicitly written to the g-buffer, it is also possible to avoid storing a value altogether. This can be done by sampling the depth stencil buffer used when rendering the g-buffer pass. A depth buffer will store the post-projection Z value divided by the post-projection W value, where W is equal to the view-space Z component of the surface position. This makes the value initially unsuitable for our needs, but fortunately it's possible to recover the view-space Z component using the parameters of the perspective projection(Baker). Once we do that, we can convert it to a normalized depth value and proceed with the same approach outlined above. However, this is unnecessary. Instead of extrapolating the view ray to the far clip plane, if we instead clamp it to the plane at Z = 1 we can scale it by the view-space Z without having to manipulate it first. Essentially, we scale the view ray in the vertex shader, rather than scaling the Z value in the pixel shader, which saves us an extra pixel shader math operation.

```
// Light vertex shader
#if VOLUMES
      // Calculate the view space vertex position
      output.PositionVS = mul(input.PositionOS, WorldMatrix);
#elif QUADS
      // For a directional light we can clamp in the vertex shades since we
      // only interpolate in the XY direction
      float3 positionVS = mul(input.PositionOS, InvProjMatrix);
      output.ViewRay = float3(positionVS.xy / positionVS.z, 1.0f);
#endif
```

```
// Light Pixel shader
#if VOLUMES
      // Clamp the view space position to the plane at Z = 1
      float3 viewRay = float3(input.PositionVS.xy / input.PositionVS.z., 1.0f);
#elif QUADS
      // For a directional light we already clamped in the vertex shader
      float3 viewRay = input.ViewRay.xyz;
#endif

// Calculate our projection constants (this can be done in the application code
// and passed in a constant buffer)
ProjectionA = FarClipDistance / (FarClipDistance - NearClipDistance);
Projection!? = (-FarClipDistance * NearClipDistance)
                              / (FarClipDistance - NearClipDistance);

// Sample the depth and convert to linear view space Z
float depth = DepthTexture.Sample(PointSampler, texCoord).x;
float linearDepth = Projection!? / (depth - ProjectionA);
float3 positionVS = viewRay * linearDepth;
```

Listing 11.12. Shader code for view space position reconstruction using a depth-stencil buffer.

If it's necessary to work in an arbitrary coordinate space, the linear depth value can be used in conjunction with the first technique to reconstruct position. This can be done by projecting the view ray onto the camera's local Z axis and using the result to figure out a proper scaling value.

```
// -- Light Pixel Shader --
// Normalize the view ray
float3 viewRay = normalize(input. ViewRay);

// Sample the depth buffer and convert it to linear depth
float depth = DepthTexture.Sample(PointSampler, texCoord).x;
float linearDepth = ProjectionB / (depth - ProjectionA);

// Project the view ray onto the camera's z-axis
float viewZProj = dot(EyeZAxis, viewRay);

// Scale the view ray by the ratio of the linear z value to the projected
// view / ray
float3 positionWS = CameraPositionWS + viewRay * (linearDepth / viewZProj);
```

Listing 11.13. Shader code for position reconstruction from a depth-stencil buffer using an arbitrary coordinate space.

When choosing a method for recovering position data, careful attention should be paid to precision and error resulting from the storage format used. The *Z/W* value stored

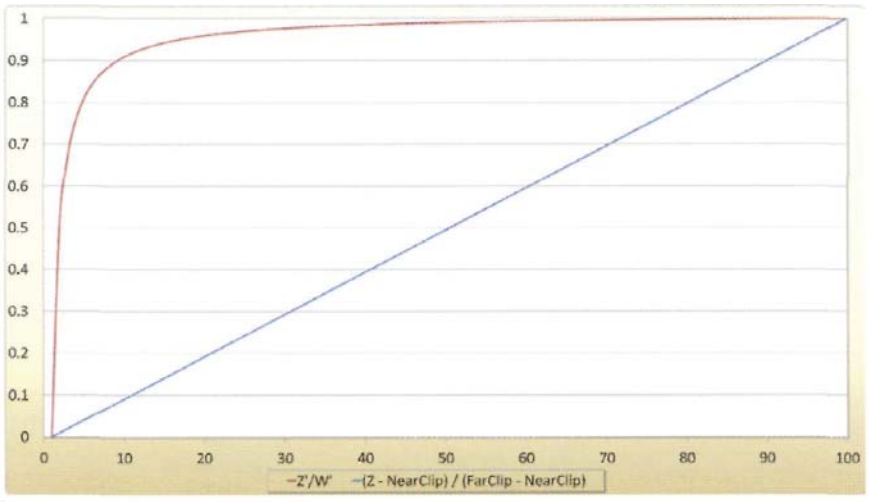

Figure 11.8. Post-perspective *z/w* vs. view space z.

in a depth buffer is non-linear with respect to the distance from the camera, and increases exponentially. Figure 11.8 shows a graph of *Z/W for* a range of *Z* values between the near clipping plane and far clipping plane (1 and 100 in this case).

Because of this non-linearity, more precision is used for depth ranges closer to the near clipping plane than to the far clipping plane, as demonstrated by the fast rise in the *Z/W* plot in Figure 11.8. This can cause error by causing reconstructed positions to become high for surfaces that are close to the far clipping plane. It should be noted that D3D11 has 32-bit floating point depth stencil formats available. However floating point numbers also have an inherent non-linear distribution of precision that causes error to increase as values move away from zero(Goldberg, 1991), which can actually compound the errors. To counteract this, it is possible to flip the distances for the near and far plane when creating a perspective projection(Persson, A couple of notes about Z, 2009), (Kemen, 2009). With such a projection *Z/W* will begin at 1.0 at the near clipping plane and decrease to 0.0, causing the two non-linear precision distributions to mostly balance out. It should be noted that if this technique is used, the direction of the depth test will also need to be reversed.

11.4.2 Light Shading Optimizations

As mentioned in the previous sections, applying lighting in a deferred Tenderer requires sampling the g-buffer textures, performing lighting calculations in a pixel shader, and summing the results with the contents of the render target. Consequently, the performance of a particular scene is often directly tied to the number of pixels shaded during the lighting pass. This

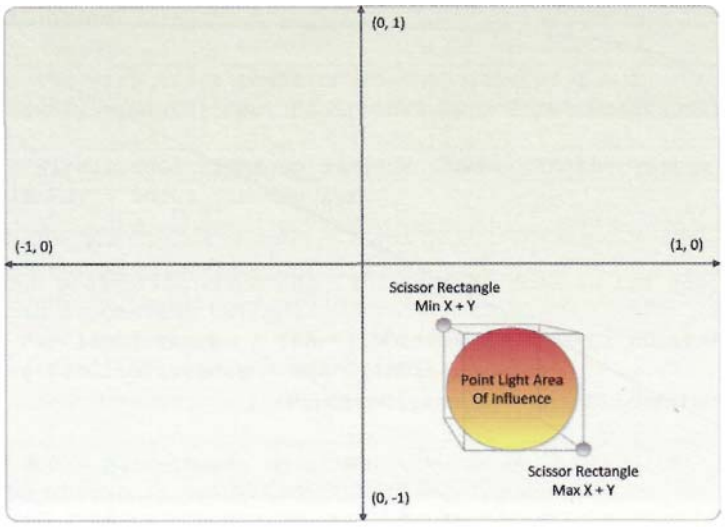

Figure 11.9. Scissor rectangle calculation for a point light.

section will focus on optimizations that can reduce the number of pixels shaded, to minimize the amount of bandwidth and/or shader ALU resources consumed by the lighting pass.

Scissor Test

The rasterizer stage of the pipeline can perform what is known as the *scissor test.*[9] This test checks whether the screen space position of each rasterized pixel intersects with a "scissor" rectangle and avoids shading any pixels that fail the test. This can be leveraged by a deferred renderer as a coarse-grained mechanism for culling pixels that won't be affected by a particular light. However, it should be noted that since this technique is based on the screen-space coverage of a light source, it only works for local light sources (point lights and spot lights), and not for global light sources (directional lights).

To effectively use the scissor test, we must calculate a screen-space bounding rectangle for the light source. Before we can do this however, we must determine the area of the screen that will be affected by a light source, which can be calculated based on the light's parameters, the camera position and orientation, and the projection parameters. This process is simplest for a point light source. Since a point light contributes equally in all directions, we can assume that the light's area of effect is perfectly spherical in both world space and view space. Thus, we can trivially fit a bounding sphere in view space to the light's area of effect. Once this bounding sphere is determined, an axis-aligned bounding box can be fitted to encapsulate the sphere. The corners of the screen-space bounding rectangle can then be determined by transforming the eight corners of the box by the projection matrix,

[9] The use of the scissor test is described in the rasterizer section of Chapter 3.

and finding the min and max X and *Y* values of the eight resulting coordinates. Figure 11.9 illustrates this process.

This process can be optimized if it is assumed that the projection is perspective, and that it is symmetrical with respect to the camera's *X* and Y axes. In this case, we can determine if the minimal or maximal *X* or *Y* value will be on the face of the box closest to the camera, or on the face of the box farthest from the camera, based on the view-space *X* and Y coordinates of the sphere. The code in Listing 11.14 uses this optimization to determine the final screen-space rectangle that will be passed to the ID3D11DeviceContext::RSSet ScissorRects method.

```
D3D11_RECT ViewLights::CalcScissorRect( const Vector3f& lightPos,
                                        float lightRange  )
{
    // Create a bounding sphere for the light., based on the position
    // and range
    Vector4f centerWS = Vector4f( lightPos, 1.0f );
    float radius = lightRange;

    // Transform the sphere center to view space
    Vector4f centerVS = ViewMatrix * centerWS;

    // Figure out the four points at the top, bottom, left, and
    // right of the sphere
    Vector4f topVS = centerVS + Vector4f( 0.0f, radius, 0.0f, 0.0f );
    Vector4f bottomVS = centerVS - Vector4f( 0.0f, radius, 0.0f, 0.0f );
    Vector4f leftVS = centerVS - Vector4f( radius, 0.0f, 0.0f, 0.0f );
    Vector4f rightVS = centerVS + Vector4f( radius, 0.0f, 0.0f, 0.0f );

    // Figure out whether we want to use the top and right from quad
    // tangent to the front of the sphere, or the back of the sphere
    leftVS.z = leftVS.x < 0.0f ? leftVS.z - radius : leftVS.z + radius;
    rightVS.z = rightVS.x < 0.0f ? rightVS.z + radius : rightVS.z - radius;
    topVS.z = topVS.y < 0.0f ? topVS.z + radius : topVS.z - radius;
    bottomVS.z = bottomVS.y < 0.0f ? bottomVS.z - radius
                                   : bottomVS.z + radius;

    // Clamp the z coordinate to the clip planes
    leftVS.z = Clamp( leftVS.z, m_fNearClip, m_fFarClip );
    rightVS.z = Clamp( rightVS.z, m_fNearClip, m_fFarClip );
    topVS.z = Clamp( topVS.z, m_fNearClip, m_fFarClip );
    bottomVS.z = Clamp( bottomVS.z, m_fNearClip, m_fFarClip );

    // Figure out the rectangle in clip-space by applying the
    // perspective transform. We assume that the perspective
    // transform is symmetrical with respect to X and Y.
    float rectLeftCS = leftVS.x * ProjMatrix( 0, 0 ) / leftVS.z;
    float rectRightCS = rightVS.x * ProjHatrix( 0, 0 ) / rightVS.z;
    float rectTopCS = topVS.y * ProjMatrix( 1, 1 ) / topVS.z;
    float rectBottomCS = bottomVS.y * ProjMatrix( 1, 1 ) / bottomVS.z;
```

```
    // Clamp the rectangle to the screen extents
    rectTopCS = Clamp( rectTopCS, -1.0f, 1.0f );
    rectBottomCS = Clamp( rectBottomCS, -l.0f, l.0f );
    rectLeftCS = Clamp( rectLeftCS, -l.0f, l.ef );
    rectRightCS = Clamp( rectRightCS, -l.0f, 1.0f );

    // Now we convert to screen coordinates by applying the
    // viewport transform
    float rectTopSS = rectTopCS * 0.5f + 0.5f;
    float rectBottomSS = rectBottomCS * 0.5f + 0.5f;
    float rectLeftSS = rectLeftCS * 0.5f + 0.5f;
    float rectRightSS = rectRightCS * 0.5f + 0.5f;

    rectTopSS = 1.0f - rectTopSS;
    rectBottomSS = l.0f - rectBottomSS;

    rectTopSS *= m_uVPHeight;
    rectBottomSS *= m_uVPHeight;
    rectLeftSS *= m_uVPWidth;
    rectRightSS *= m_uVPWidth;

    // Final step is to convert to integers and fill out the
    // D3D11_RECT structure
    D3D11_RECT rect;
    rect.left = static_cast<LONG>( rectLeftSS );
    rect.right = static_cast<LON6>( rectRightSS );
    rect.top = static_cast<LONG>( rectTopSS );
    rect.bottom = static_cast<LONG>( rectBottomSS );

    // Clamp to the viewport size
    rect.left = max( rect.left, 0 );
    rect.top = max( rect.top, 0 );
    rect.right = min( rect.right, static_cast<LONG>( m_uVPWidth ) );
    rect.bottom = min( rect.bottom, static_cast<LONG>( m_uVPHeight ) );

    return rect;
}
```

Listing 11.14. C++ code for fitting scissor rectangle to a point light.

For a spot light, we can use the same process of fitting a bounding sphere and determining the screen-space extents. The code in Listing 11.14 will work for a spot light without any modifications, since the resulting bounding sphere will completely encompass the conical area of effect, regardless of the light's orientation and angular attenuation parameters. If necessary, the angular attenuation factors can be used to fit a bounding cone to the light, to which a tighter bounding sphere can be fit.

An alternative to this approach is to approximate the light's bounding cone using a mesh of vertices, and then calculate the bounding box, based on the vertices' projected positions in screen space. This requires more calculations at runtime, but it can potentially result in a tighter fit than using a bounding sphere.

GPU-Generated Quads

The algorithm used for generating the extents of a scissor rectangle can be trivially adapted for use in a vertex shader or a geometry shader. This effectively offloads the computation to the GPU, and removes the need for a state change before drawing each light source. The removal of the state change allows multiple light sources to be batched together into a single draw call, which can dramatically reduce CPU overhead for large numbers of lights. See the light prepass sample for a demonstration of this approach.

Bounding Geometry

As mentioned in the previous section, it's possible to determine bounding geometry for point and spot lights, based on their position and attenuation properties, for use with the scissor test. It's possible to take this concept a step further, and create a triangle mesh representing a light's bounding volume and then render that, instead of using a full screen quad. This frees us from the limitations of screen-space rectangles and also makes natural use of the CPU's vertex processing and rasterization capabilities.

To generate the geometry, we do not need to take into account an individual light's properties. Instead, we can generate a single sphere and a single cone to be used with all lights, and then apply a world transform by calculating translation, rotation, and scale components based on the light properties. For a point light, it is simplest to create the sphere mesh with a radius of 1 and centered at the origin. Then our translation can be set to the light's position, and the *X*, *Y*, and Z scales can all be set to the light's effective range. For a spot light, a cone aligned with the Z axis (with the tip at the origin) and an angle of 45 degrees should be used. Then the translation and orientation can be set to the light's position and orientation, and the Z scale can be set to the effective range. The *X* and *Y* scales can then be set by determining the radius of the cone, which is done by calculating the tangent of the spotlight angle and multiplying with the range.

When the bounding geometry is rendered, we must also choose whether the pixels should be rendered with front-face or back-face culling. We can't simply disable culling, because pixels would get lit twice. This decision must be carefully made, because if the bounding geometry intersects with the near clipping plane or the far clipping plane of the camera, some of the front-facing or back-facing triangles will not be rasterized. Thus, if a bounding volume intersects with the near clipping plane, back-face culling should be used. If it intersects with the far-clipping plane, back-face culling should be used. If it intersects with both, this means the light is very large and should be rendered with a quad or some other method. As an alternative to switching culling modes, the vertices can also be clamped to the near and far clipping planes in the vertex shader. This can be useful if lights need to be batched together into a single draw call.

Aside from also limiting pixel shading to the bounding volume's screen space projection, using bounding geometry also allows us to make use of hardware depth testing to cull

additional pixels. The idea is that we only want to light pixels where the scene geometry intersects with our light bounding meshes. This means that we want to reject pixels where the light is occluded by scene geometry, as well as pixels where the light is "floating in air." When rendering back-face geometry, we can reject the first case by setting the depth test to D3D11_COMPARISON_LESS_EQUAL as our depth comparison mode. For rendering front-face geometry, we can reject the second case by using D3D11_COMPARISON_GREATER_EQUAL. Since we must use either front-face or back-face culling, we can't reject both cases simultaneously in a single pass. As such, it can be advantageous to use a heuristic to estimate which culling/depth testing mode would result in more pixels being culled.

Stencil Test

The stencil buffer lets us store a unique 8-bit integer value per pixel, which we can use in simple logical tests that determine whether or not a pixel should be culled. With a deferred renderer, we can use the stencil buffer to store a mask that indicates which geometry needs to be lit, and thus avoid shading pixels that don't require lighting. This can result in large savings if a significant portion of the screen contains a *skybox*[10] or background geometry, or if there is a lot of scene geometry with materials that don't require dynamic lighting.

To implement this, we simply set the stencil comparison function to D3D11_COMPARISON_ALWAYS for lit geometry while filling the g-buffer, which causes the reference value to be written to the stencil buffer for each pixel rasterized. For the reference value, we use some known value that is different than what the stencil buffer was cleared to. Then when rendering the lights, we use the same reference value and set the test to D3D11_COMPARISON_EQUAL. This causes only pixels with the correct stencil value to be written, and any others to be culled. Since most modern graphics hardware features Hi-Z units that can perform Z-culling and stencil culling before the pixel shader is executed, the stencil test can effectively prevent pixels from being shaded at all.

This concept can also be extended to implement *light masking*. Light masking allows certain lights to only affect certain scene geometry. For example, a set of lights could be used to only light a player character, and not the level geometry. Or if a light is in a room, it can be made to only affect the walls of that room, without relying on shadowing or attenuation to keep it from "bleeding through" into adjacent rooms. Since a stencil buffer has eight bits that can be tested independently, the logical way to implement light masks is to have each bit represent a *light group* that a light or object in the scene can belong to. Essentially, a light will only affect geometry that belongs to the same group. So in the case of adjacent rooms, each room would have a unique light group that the lights and geometry

[10] A *skybox* is a cube of geometry rendered with the camera at the center of the cube. A cube texture is typically applied to it, giving the appearance of geometry very far from the viewer. Since it represents objects that are very far away, it would not receive lighting of this sort and can hence be eliminated.

would belong to. This way the light source doesn't "bleed through" the wall, and only affects the room it is in.

When rendering the scene geometry, we can use the same premise as above, except that we set the reference value to an 8-bit unsigned integer created by bitwise OR'ing all of the light groups to which it belongs. Then during the lighting pass, we also determine the same 8-bit value using the light groups that a light belongs to. However, instead of using that value as the reference, we set it as the stencil read mask. This mask is bitwise AND'ed with the value in the stencil buffer, and the reference value is compared with the stencil buffer value, using the specified test function. Thus, if we use a reference value of 0 and D3D11_COMPARISON_LESS as the test function, the stencil test will pass if the geometry at that pixel and the light both belong to any of the same light groups.

Screen Space Tile Sorting

This technique is another exploitation of the spatial coherence of local light sources. The basic concept involves splitting the screen into tiles of a fixed size and determining which light sources need to be rendered for each tile (Balestra & Engstad, 2008), (Lauritzen, 2010). Then the lights are all rendered in one batch or a series of batches, where the pixel shader evaluates and sums the lighting contribution for all lights in the batch. The diagram in Figure 11.10 illustrates the concept.

At first glance, it may seem that this technique is inferior to the other techniques mentioned earlier, since pixels can only be culled at the level of granularity used for the tiles. This can be especially inefficient when a small light intersects with multiple tiles. However, this technique has three key advantages. The first is that the g-buffer only needs to be sampled once for each batch, which potentially reduces the bandwidth required for rendering each light, as long as the batches contain multiple lights. This also means that the g-buffer's attributes only have to be unpacked or reconstructed once per batch, which saves shader ALU cycles. The second is that the lighting contribution only has to be written to the render target (and possibly blended) once per batch, which can further reduce the bandwidth usage of the lighting pass. The third is that it makes the lighting computations suitable for evaluation on general-purpose computation resources such as CPUs or compute shaders (Lauritzen, 2010), since no rasterization is required and the memory access is extremely coherent.

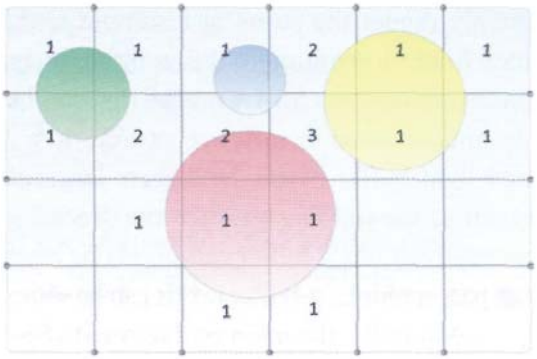

Figure 11.10. Tiled light binning.

If this technique is performed using the standard GPU rendering pipeline, it

can be combined with some of the other optimizations mentioned earlier in this section. For instance, depth testing can still be used by rendering a quad and setting the view-space Z of each vertex to the maximum or minimum depth of all lights in the batch. In addition, the scissor test can also still be used to cull pixels for smaller lights.

11.5 Anti-Aliasing

Deferred rendering has many advantages, which result from the way it reorganizes the rendering pipeline. However at the same time, this reorganization makes it incompatible with certain techniques and enhancements that have previously worked with traditional forward rendering. One of the biggest examples is multisample anti-aliasing, or MSAA[11] MSAA works by altering the rasterization process so that triangle coverage is computed at a subpixel level, so that the output of each pixel can be written to none, some, or all of the subsamples in the render target. The resulting subsamples are then blended to produce the final output. For this to work, we must be rasterizing the scene geometry while rendering the final pixels. Also, for those pixels where geometry intersects, the subsamples must contain the proper color values that would result from fully lighting and shading those triangles. With classic deferred rendering, we satisfy neither of those conditions, while with light prepass rendering, we only satisfy one. In this chapter we will discuss alternatives and workarounds for providing anti-aliasing in a deferred Tenderer.

11.5.1 Supersampled Anti-Aliasing

Supersampled anti-aliasing, commonly abbreviated *SSAA,* is the simplest alternative type of anti-aliasing to implement, and was mentioned briefly in Chapter 3. With SSAA, we simply render the scene at a sample rate (resolution) that is higher than the output, and then filter (downsample) the resulting image before presenting it to the display. Some GPU hardware features SSAA modes that can be toggled in the driver settings, but it can be easily implemented in software as well. In a deferred Tenderer, we simply make the g-buffer and light buffer be render targets larger than the back buffer, and then downsample the result of the lighting pass (or the second geometry pass, in the case of light prepass). Or alternatively, the g-buffer can simply be rendered at a higher resolution. Then, during the lighting pass multiple, g-buffer texels can be sampled and evaluated, so that the result can be filtered.

Naturally, the main problem with SSAA is performance. Increasing the resolution by a factor of 2 or 4 puts additional pressure on nearly every part of the GPU pipeline, and also greatly increases the amount of GPU memory (and corresponding bandwidth) consumed

[11] For more details on MSAA, see Chapter 3, "The Rendering Pipeline.

by render targets. However, SSAA is certainly effective, and it may be worthwhile to provide as an option if no other type of AA is implemented.

11.5.2 Multisampled Anti-Aliasing

As mentioned previously, breaking the rendering pipeline into 2 or 3 passes makes it incompatible with MSAA. However, we can still use hardware support for MSAA to produce an antialiased output, without having to resort to supersampling.

Let's start by analyzing the g-buffer pass. If we make our g-buffer render targets multisampled and enable MSAA while rendering them, this gives us a multisampled description of the surface and material properties. This means that for pixels where a triangle doesn't have complete coverage, the g-buffer properties for all triangles that were rasterized will be contained in the individual subsamples. Thus, if we sample and light all of these subsamples individually and then filter or average the results, we would produce an antialiased image. A sample function for performing this is provided in Listing 11.15.

```
// Textures
Texture2DMS<float4> NormalTexture               : register( t0 );
Texture2DMS<float4> DiffuseAlbedoTexture        : register( t1 );
Texture2DMS<float4> SpecularAlbedoTexture       : register( t2 );
Texture2DMS<float4> PositionTexture             : register( t3 );

// Helper function for extracting G-Buffer attributes
void GetGBufferAttributes( in float2 screenPos,
                           in int sslndex,
                           out float3 normal,
                           out float3 position,
                           out float3 diffuseAlbedo,
                           out float3 specularAlbedo,
                           out float specularPower )
{
      // Determine our indices for sampling the texture based
      // on the current screen position
      int2 samplePos = int2( screenPos.xy );

      normal = NormalTexture.Load( samplePos, sslndex).xyz;
      position = PositionTexture.Load( samplePos, sslndex).xyz;
      diffuseAlbedo = DiffuseAlbedoTexture.Load(samplePos, sslndex).xyz;
      float4 spec = SpecularAlbedoTexture.Load(samplePos, sslndex);

      specularAlbedo = spec.xyz;
      specularPower = spec.w;
}

// Lighting pixel shader that performs MSAA resolve
float4 PSMain( in float4 screenPos : SVPosition ) : SVTargete
```

```
{
        PSOutput output;
        float3 lighting = 0;

        for ( int i = 0; i < NUMSUBSAMPLES; ++i )
        {
                float3 normal;
                float3 position;
                float3 diffuseAlbedo;
                float3 specularAlbedo;
                float specular-Power;

                GetGBufferAttributes( screenPos.xy, i, normal, position,
                                      diffuseAlbedo, specularAlbedo,
                                      specularPower );
                lighting += CalcLighting( normal, position, diffuseAlbedo,
                                          specularAlbedo, specularPower );
        }

        lighting /= NUMSUBSAMPLES;

        return float4( lighting, 1.0f );
}
```

Listing 11.15. Pixel shader code for lighting MSAA subsamples.

One issue we must be aware of with this approach is that it is not always desirable to sample all subsamples from the g-buffer and compute lighting for them. This is the case when the geometry rendered for the lighting doesn't fully cover a pixel, or when the depth/stencil tests don't pass for all subsamples. To ensure that only the appropriate samples are used, we can take SV_Coverage as an input to the pixel shader. This semantic provides a `uint` value, in which each bit corresponds to an MSAA sample. A bit with a value of 1 indicates that the triangle coverage test passed for the corresponding MSAA sample point, while a value of 0 indicates that it failed. If we perform a bitwise AND of this mask with a bit value for the current sample in the loop, we can determine whether we should skip the subsample. The code in Listing 11.16 demonstrates how the code from Listing 11.15 can be modified to follow this approach.

```
float4 PSMain( in float4 screenPos : SV_Position,
in uint coverageMask : SV_Coverage ) : SV_Target0
{
        PSOutput output;
        float3 lighting = 0;
        float numSamplesApplied = 0.0f;
        for ( int i = 0; i < NUMSUBSAMPLES; ++i )
        {
                if ( coverageMask & ( 1 << i ) )
```

```
            {
                float3 normal;
                float3 position;
                float3 diffuseAlbedo;
                float3 specularAlbedo;
                float specularPower;

                GetGBufferAttributes( screenPos.xy, i, normal,
                                      position, diffuseAlbedo,
                                      specularAlbedo, specularPower );
                lighting += CalcLighting( normal, position,
                                          diffuseAlbedo, specularAlbedo,
                                          specularPower );

                ++numSamplesApplied;
            }
        }

        lighting /= numSamplesApplied;

        return float4( lighting, 1.0f );
}
```

Listing 11.16. Pixel shader code for lighting MSAA subsamples with coverage test.

Another way to ensure proper g-buffer sampling is to run the pixel shader at per-sample frequency instead of at per-pixel frequency. In this case, the pixel shader only runs if the triangle coverage test and the depth/stencil test both passed for a given sample, so there is no need to check the input coverage mask.

Using MSAA gives us better performance than supersampling, because we don't have to render the g-buffer at a higher resolution. However, we can do even better. When rendering with MSAA, the majority of the pixels on the screen will be fully covered by a triangle from the scene geometry. Consequently, all of the subsamples of a given pixel within the g-buffer will be the same. We can take advantage of this to optimize the approach outlined above by only lighting one subsample for pixels where all subsamples are identical. One possible approach is to read all subsamples of one of the g-buffer textures in the lighting pass, compare for equivalence, and then dynamically branch based on that result. While this can work in some cases, it's possible that the subsamples will be identical (or within the threshold used) for one g-buffer texture, while one or more of the other textures are different. In addition, performance will vary depending on the granularity of dynamic branching in the GPU. If one pixel takes the slow branch, where it evaluates all subsamples, this can cause all adjacent pixels within a certain tile size to take both sides of the branch.

To avoid this issue, a stencil mask can be used, rather than branching in the pixel shader. The mask can be generated with a full-screen pass that samples the g-buffer and

performs the comparison, and then uses the clip() or discard intrinsics to prevent that pixel from being output. However this can still suffer the same problems inherent with using an equality comparison on the g-buffer contents. Once the stencil mask is generated, the lighting must be rendered with two passes: one where the subsamples are lit individually and one where only one sample is used (Persson, Deferred shading 2, 2008).

Ideally, we would want to mark "edge" pixels (pixels where subsamples aren't all identical), based on the actual subpixel triangle coverage, rather than performing a comparison after the fact. In Direct3D 11 the pixel shader can get this information directly through the SV_Coverage semantic, making it trivial to determine if a pixel is fully covered, and then output an appropriate value to the g-buffer. The code in Listing 11.17 demonstrates how this can be implemented.

```
// G-Buffer Pixel shader that generates an MSAA mask using SV_Coverage
float4 PSMain( in PSInput input, in uint coverageMask : SV_Coverage ) : SV_Target0
{
      // Compare the input mask with a "full" mask to determine if all
      // samples passed the coverage test
      const uint FullMask = ( 1 << NUMSUBSAMPLES ) - 1;
      float edgePixel = coverageMask != FullMask ? 1.0f : 0.0f;


}
```

Listing 11.17. Shader code for edge mask generation using SV_Coverage.

MSAA with Light Pre-Pass Deferred Rendering

With light pre-pass deferred rendering, our rendering pipeline is split into three steps, instead of two. Thus, if the MSAA subsamples are resolved during the lighting pass and the final pass is rendered without MSAA, the albedo from one surface can incorrectly be applied to a lighting value that is the combined result of two different surfaces. Or alternatively, if MSAA is used for the final pass, the two separate albedo values will be used, but the lighting values will already have been combined. This will produce artifacts, since the lighting equation is nonlinear.

Ideally, the lighting pass should calculate lighting for each individual subsample in the g-buffer and then store the results to the corresponding subsample of an MSAA render target. While Direct3D 11 allows loading the subsamples of a texture, it doesn't have a mechanism for specifying separate values for the individual subsamples of a render target when a pixel shader runs at a per-pixel frequency. To work around this, we can have the lighting pixel shader run at the per-sample frequency, so that the light values for each subsample are preserved. The shader code in Listing 11.18 demonstrates this approach.

```
// Light pixel shader that executes once per MSAA subsample
float4 PSMainPerSample( in float4 screenPos : SV_Position,
in uint subSampleIndex : SV_SampleIndex )  : SV_Target0
{
      float3 normal;
      float3 position;
      float specularPower;

      // Get the G-Buffer values for the current sub-sample, and calculate
      // the lighting
      GetGBufferAttributes( screenPos.xy, ViewRay, subSampleIndex,
                                    normal, position, specularPower );

      return CalcLighting( normal, position, specularPower );
}
```

Listing 11.18. Per-sample light buffer pixel shader code.

Naturally, running pixel shaders at per-sample frequency incurs a higher overhead than running them per-pixel. To mitigate this effect, the stencil mask technique mentioned earlier can be used to mask off pixels that don't need to be shaded per-sample.

For the final pass, we no longer need to output separate values per-sample. Instead, we want to sample the light buffer for all relevant subsamples, filter the results, and finally apply the albedo terms. Thus, running the pixel shader at per-pixel frequency and relying on the coverage and depth/stencil tests is perfectly adequate. However, we must be careful not to inadvertently apply lighting from subsamples not covered by the triangle currently being rasterized, or artifacts will occur. To avoid these artifacts, we can make use of the input coverage mask, much like the lighting pass of a classic deferred Tenderer. The code in Listing 11.19 demonstrates how to do this.

```
float4 PSMain( in PSInput input, in uint coverageMask : SV_Coverage ) : SV_Target0
{
      // Determine our coordinate for sampling the texture based on
      // the current screen position
      int2 sampleCoord = int2( input.ScreenPos.xy );

      float3 diffuse = 0;
      float3 specular = 0;
      float numSamplesApplied = 0.0f;
      // Loop through the MSAA samples and modulate diffuse and specular
      // lighting by the albedo
      for ( uint i = 0; i < NUMSUBSAMPLES; ++i )
      {
            // We only want to apply a lighting sample if the geometry we've
            // rasterized passed the coverage test for the corresponding
            // MSAA sample point
            if ( coverageMask & ( 1 << i ) )
```

```
                {
                        // Sample the light target for the current sub-sample
                        float4 lighting = LightTexture.Load( sampleCoord, i );

                        // Seperate into diffuse and specular components
                        diffuse += lighting.xyz;
                        specular += lighting.www;

                        ++numSamplesApplied;
                }
        }

        // Apply the diffuse normalization factor
        const float DiffuseNormalizationFactor = 1.0f / 3.14159265f;
        diffuse *= DiffuseNormalizationFactor;

        // Apply the albedos
        float3 diffuseAlbedo = DiffuseMap.Sample( AnisoSampler, input.TexCoord ).rgb;
        float3 specularAlbedo = float3( 0.7f, 0.7f, 0.7f );
        diffuse *= diffuseAlbedo;
        specular *= specularAlbedo;

        // Final output is the sum of diffuse + specular divided by number of
        // samples covered
        float3 output = ( diffuse + specular ) / numSamplesApplied;
        return float4(output , 1.0f );
}
```

Listing 11.19. Final pass pixel shader code for light prepass deferred rendering with MSAA.

As with the lighting pass, it's possible to use a stencil mask or dynamic branching to prevent sampling and applying multiple subsamples from the light buffer. However we should keep in mind that the work done per-subsample is very little in the final pass, and thus the savings in terms of pixel shading performance are much smaller than in the lighting pass. Also, if a stencil mask is used, this means rendering all of the scene geometry twice, which makes it very likely that performance could actually be worse than if we simply looped through all of the subsamples. Either way, profiling should be done on the target hardware to determine which approach produces the best performance.

11.5.3 Screen-Space Anti-Aliasing

In this section, we'll describe techniques that apply an anti-aliasing effect in screen space after the scene has been rendered and shaded. The main advantage of such techniques is that they are mostly orthogonal to the techniques used to render the geometry, which

makes them suitable not just for forward rendering, but also for deferred rendering, or any non-traditional rendering pipelines. They can also be quite a bit cheaper than MSAA, since MSAA can greatly increase the bandwidth usage and memory footprint (especially when deferred rendering is used). This also helps screen-space techniques scale better with higher resolutions. One added benefit from these techniques is that they can be applied after HDR tone mapping. When an MSAA resolve happens before tone mapping, the quality often suffers in high-contrast areas, because tone mapping uses a non-linear operator. This manifests as triangle edges that appear to have no anti-aliasing applied to them (Persson, Post-tonemapping resolve for high quality HDR anti-aliasing in D3D10, 2008). Moving the resolve to after the tone mapping step can greatly reduce the artifacts that occur in high contrast areas.

The primary disadvantage of these techniques is that they typically don't have any sort of subpixel information available to them, which makes it difficult or impossible to reconstruct signals for small or thin geometry. It also means that geometry edges can't snap to subpixel positions, as they can with MSAA, which makes it difficult to alleviate temporal aliasing artifacts.

Edge Blur

One of the oldest and simplest implementations of screen-space AA involves detecting "edges" in the scene and applying a blur to the pixels (Policarpo & Fonseca, 2005). Typically, an edge-detection operator such as a Sobel filter is used to detect edges in screen space, with the detection applied to scene depth and normal vectors. Using normal vectors and depth, rather than color or luminosity, restricts edge detection to triangle edges, preventing triangle interiors from inadvertently being blurred. Only depth is required to detect edges at mesh silhouettes, while use of normal vectors is required to detect edges where triangles intersect. However, even with normal vectors, this approach will still fail for coplanar triangles that have different material properties.

The obvious downside to this technique is that quality is generally rather poor. Applying a box filter or Gaussian filter to pixel colors can reduce aliasing artifacts somewhat, but it can also destroy detail in the process.

Morphological Anti-Aliasing

Morphological anti-aliasing (Reshetov, 2009), commonly abbreviated as *MLAA,* is another screen-space technique that uses sophisticated pattern recognition techniques to identify triangle edges. Rather than using a simple edge detection operator like Sobel that operates on a single pixel at a time, the algorithm works by splitting the screen into tiles and detecting all edge patterns within a tile. These patterns are then used to estimate the actual triangle coverage for a pixel, which is then used to blend the colors on both sides of the edges. Since the algorithm reconstructs the actual triangle edges and doesn't just rely on a

blur, the results can be excellent for still images. However moving images can still suffer from temporal artifacts (due to the fundamental limitation of working in screen space with no subpixel information), although those are generally reduced compared to not having any anti-aliasing at all. The algorithm will also fail frequently on thin, wire-like geometry, where the resulting aliasing patterns are very low frequency.

The original reference implementation provided by Intel runs entirely on the CPU, using streaming SIMD instructions. Using the CPU can reduce the GPU load and can be an effective use of otherwise idle CPU cores; however, the cost of transferring render target data to and from the GPU can significantly reduce overall performance. Instead, it is likely to be more desirable to keep the implementation entirely on the GPU, which can be done using only pixel shaders (Biri, Herubel, & Deverly, 2010). A compute shader implementation is also possible and could potentially benefit from the added flexibility and shared memory resources.

11.6 Transparency

The concept of deferred rendering works extremely well for opaque geometry, since we can make the assumption that for each pixel (or subsample, in the case of MSAA), only one surface is visible. This allows us to use a g-buffer to store the surface information, since we have one g-buffer texel available for each shaded pixel. But transparent geometry causes a problem with this approach, since it breaks our assumption of 1 surface per pixel. Instead we have 0 or 1 opaque surfaces visible, with an unbounded number of transparent surfaces layered on top of it. Because of this, we have to deal with transparency as a special case when using deferred rendering.

11.6.1 Forward Rendering

The simplest way to handle transparent geometry is to fall back to forward rendering. The opaque geometry can be rendered and fully shaded using deferred techniques, and afterwards, the transparent geometry can be forward rendered and blended right on top of it. For simple transparent materials that don't require dynamic lighting or shadowing, this approach works well, without any major problems. However, if dynamic lighting is required, it essentially means that a full forward rendering pipeline has to be implemented side by side with the deferred pipeline. This negates one of the biggest advantages of deferred rendering, which is simplifying the rendering implementation. In fact, with this case, we end up with a Tenderer that is *more* complex than a traditional forward Tenderer, since two separate pipelines are needed. The transparent geometry will also suffer from all of the

usual forward-rendering performance drawbacks that we discussed in the beginning of this chapter, such as poor efficiency for small triangles, and reduced batching.

Due to the downsides associated with using dynamic lighting with transparent materials, it's desirable to use simplified materials as much as possible. For many glass-like materials, dynamic lighting and shadowing don't provide the most important visual cues required for realism. Typically it's more important to provide reflections, which can be implemented with static reflection maps. The overall ambient or diffuse lighting term can also be precomputed and stored in vertices or *lightmaps,*[12] which allows the transparent geometry to blend in with the opaque geometry.

11.6.2 Multiple G-Buffer Layers

A natural extension of deferred rendering to support transparent geometry is to use multiple g-buffers to store surface information. Having N g-buffers allows up to N surfaces to be independently lit and blended using standard deferred lighting techniques, an arrangement suitable for use with overlapping transparent geometry. The major downside is that additional memory and bandwidth are required to store and write out the multiple g-buffer layers.

Coarse Binning

To fill the g-buffer layers, a simple approach is to do a *coarse binning* of transparent geometry on the CPU. With this case, the first layer is reserved for opaque surfaces, and the transparent objects are binned into the rest of the layers. Which layer an object is sent to can be determined by reserving depth ranges for certain layers, or by attempting to determine where objects overlap by using simple bounding volumes. With both approaches, it is impossible to properly handle the case of intersecting triangles, which is a fundamental limitation of order-dependent transparency methods. Then, during the lighting pass, each light can sample all of the layers and light them individually, and afterward, each layer can be blended, based on the opacity of the surface stored in the g-buffer.

Unfortunately, in this case, depth-buffer-based optimizations for light volumes won't work, since there is no longer one depth value that can be used. However if the depth is sampled in the shader for each layer, dynamic branching can be used to skip the lighting calculations if the depth is outside of the light bounds. The stencil buffer can also be used to mark pixels where a layer doesn't need to be lit, but this requires multiple passes per light. It is also possible to maintain a separate light buffer for each layer, which allows hardware depth optimizations to be applied, at the expense of more memory and an additional pass.

[12] A *lightmap* is a 2D texture that contains precomputed lighting values for triangle surfaces.

Depth Peeling

Another approach to handling the layers is to use *depth peeling* (Everitt, 2001). With depth peeling, all geometry is rendered for each layer. When rendering, the depth from the previous layer is sampled and compared with the depth of the surface being rendered. A given pixel is written only if its depth is greater than the same pixel in the previous layer. The result is that the overlapping geometry is "automatically" sorted, based on depth at a per-pixel level. Then, in the lighting pass, the layers are once again lit in order and blended together. The main advantage of this technique is that it is not dependent on the order of submission for geometry, which means that no CPU-side sorting is required. It also properly handles cases where transparent triangles overlap, since depth is sorted per-pixel. The obvious downside is that all scene geometry has to be rendered N times, where N is the number of depth layers in the scene. The added pixel shader cost can be mitigated somewhat if a depth-only prepass is performed first, which allows depth testing to be used to ensure that only one pixel per layer is shaded when writing out the g-buffer (Persson, Deep deferred shading, 2007).

11.6.3 Inferred Lighting

Inferred lighting (Kircher & Lawrance, 2009) is a technique that was initially developed to allow lighting to be run at a resolution lower than the final output resolution. It is a variation of light pre-pass deferred rendering, where the results of the lighting passes are bilaterally upsampled while rendering the scene geometry a second time. The bilateral upsample is performed by taking four samples from the g-buffer and comparing depth, normal, and object ID values. Samples that aren't within a threshold are rejected, which ensures that lighting performed for a different surface isn't applied to the geometry being rendered.

While the aim of the technique was primarily to enable lower-resolution shading as a performance optimization, it also enabled a novel method for handling transparent geometry. Since the bilateral filter automatically rejects pixels from the light buffer that aren't from the same mesh, it is possible to interleave transparent geometry into the g-buffer, using a stipple pattern. If the g-buffer is divided into a grid of 2x2 quads, a pattern can be used, where one texel is reserved for opaques while the other three provide three layers of transparent geometry. Thus, the opaque geometry would be rendered to the top-left texel, the first transparent layer would be in the top-right area, and so on. The diagram in Figure 11.11 illustrates this concept.

To assign transparent geometry to a particular layer, a coarse sorting pass can be used, as described in the "Multiple G-Buffer Layers" section. In this case, it is desirable to keep the sorting on the CPU, so that the shader has advance knowledge of which stipple pattern must be used. During the lighting pass, the lights can sample the g-buffer normally, without knowing or caring whether a texel belongs to opaque or transparent geometry. Finally, all

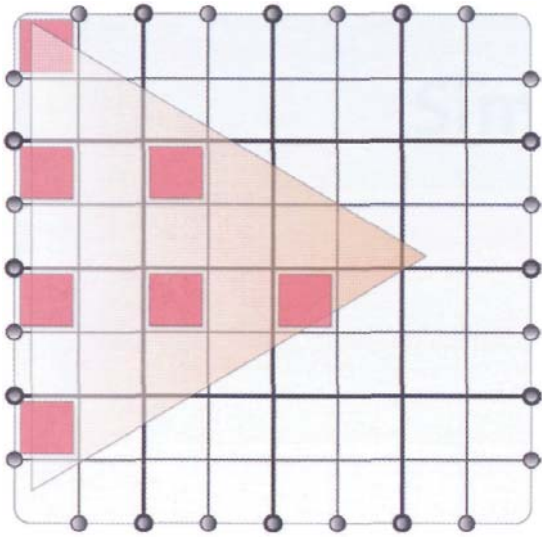

Figure 11.11. Stipple pattern for rendering transparent geometry with inferred lighting.

of the scene geometry is rendered again. The lighting buffer is sampled again, and samples from other meshes are discarded.

The result is that for quads where there is no transparent geometry, the opaque geometry is shaded at the resolution used for the g-buffer. However, for each layer of overlapping transparency, the sampling rate decreases by 25%. So in the worst case, where there are 3 layers of overlapping transparency, the opaque geometry is shaded at 1/4 rate. Transparent geometry is always shaded at 1/4 rate, due to the stipple pattern. This can cause noticeable artifacts when high frequency normal maps are used, or when shadow mapping is used. However the diffuse albedo map is still sampled at the full rate of the output resolution, preserving any high-frequency details contained in them. Aside from the sampling rate issue, there is an added performance cost from the bilateral upsample. This cost can be compensated by reducing the resolution of the g-buffer and lighting buffer; however this compounds the sampling frequency issues mentioned above.