Deferred Shading with Multiple Render Targets

Nicolas Thibieroz
PowerVR Technologies

Introduction

Traditional rendering algorithms submit geometry and immediately apply shading effects to the rasterized primitives. Complex shading effects often require multiple render passes to produce the final pixel color with the geometry submitted every pass. Deferred shading (aka quad shading) is an alternative rendering technique that submits the scene geometry only once, storing per-pixel attributes into local video memory to be used in the subsequent rendering passes. In these later passes, screen-aligned quads are rendered, and the per-pixel attributes contained in the buffer are retrieved at a 1:1 mapping ratio so that each pixel is shaded individually. The following figure illustrates the principle of deferred shading.

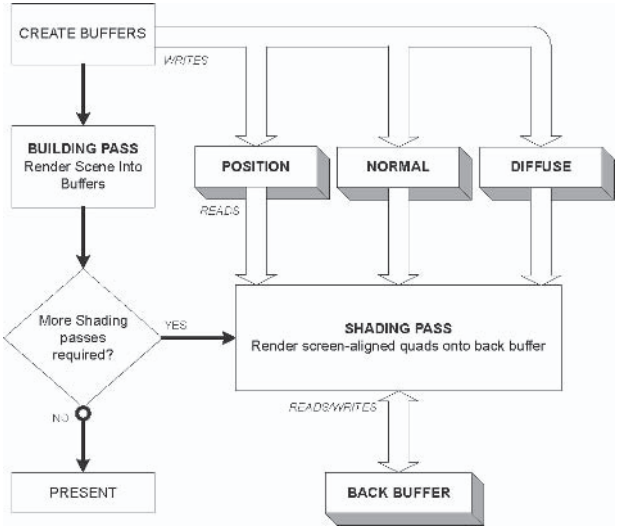

Figure 1: Deferred shading flow diagram with arbitrary examples of stored data (position, normal, and color)

Deferred shading has a number of advantages over traditional rendering. Firstly, only a single geometry pass is required, even if shadow algorithms are used. By

virtually eliminating multiple geometry passes, the saved vertex throughput can be used to dramatically increase the polygon budget of the single geometry pass, improving the scene's realism without compromising the performance of the shading passes (as they are not dependent on underlying geometry).

Secondly, all shading calculations are performed *per-pixel*, as each pixel has a unique set of properties. For shading effects that simulate lighting, this is preferable to using interpolated vertex shader outputs. This subtle difference can have a dramatic impact on the quality of the visual results.

Thirdly, deferred shading has the advantage of reducing pixel overdraw, as only the initial geometry pass may have an average overdraw value above 1. This is because all shading passes operate on pixels that are already visible; thus, no overdrawn pixel will ever be touched by the pixel shader during the shading passes.

These advantages make deferred shading an interesting alternative to traditional multi-pass rendering, though the accrued memory footprint and bandwidth requirements need careful consideration when implementing this technique. This article describes deferred shading in detail and gives practical implementation examples that show how deferred shading can be applied to current and future games using the DirectX 9 API and beyond.

Multiple Render Targets

Prior to DirectX 9, one could only output a maximum of 32 bits consisting of four color components to a render target in a single pass. However, deferred shading requires a greater number of components to accommodate the pixel attributes calculated during the initial pass. Multiple render targets (MRTs), a new feature of DirectX 9, allow up to four render targets to be written to in the same rendering pass, bringing the total number of output components to 16 and the maximum precision to 512 bits (although these can vary depending on the host 3D device). Without MRT support, outputting more than four components would require additional geometry passes.

These MRTs are used to store scene information during the geometry pass (or "building pass") and are then accessed as textures during the shading passes. Note that MRTs have the following limitations in the DirectX 9 API:

- They must be of identical size.
- They can only be of different bit depths if the `D3DPMISCCAPS_MRTINDEPENDENTBITDEPTHS` cap is exported by the 3D device.
- Dithering, alpha testing, fogging, blending, or masking are only supported if the 3D device exposes the `D3DPMISCCAPS_MRTPOSTPIXELSHADERBLENDING` cap.
- They may not be antialiased.

Each MRT contains per-pixel information about the scene and therefore should be of the same size as the main render target (the back buffer). Because the back buffer's width and height are usually not a power of two, 3D devices not supporting the `D3DPTEXTURECAPS_NONPOW2CONDITIONAL` cap will need to create MRTs at the

next power of two size above the main render target dimensions (e.g., for a 1280x1024 back buffer, the MRT's size will be 2048x1024). Although non-power of two MRTs have limitations, these do not directly affect the algorithm.

Multi-element textures (METs) are another feature, albeit less flexible, of DirectX 9 that closely resembles MRTs. METs are basically a set of textures *of the same format* packed together. This limitation and the fact that DirectX 9 exports a single MET format called `D3DFMT_MULTI2_ARGB8` (two textures of 8888 format) make METs unsuitable for deferred shading.

Attribute Buffer

The attribute buffer is the name that we give to the scene data we are storing in our MRT textures during the building pass. The choice of stored data depends on the shading model. For instance, Gouraud-shaded directional (or parallel) lights only require each pixel's normal and material properties, whereas Phong-shaded point or spotlights require the pixel's position in addition to its normal and material properties.

The implementation detailed in this article assumes all lights required during the shading passes are based on the Phong model, and therefore the attribute buffer contains the following:

- Pixel position (X, Y, Z)
- Pixel normal vector (X, Y, Z)
- Pixel diffuse color (R, G, B)

Pixel Position

This is the world space position of the pixel. Note that it is also possible to store the position in a different coordinate system (eye space being the other logical alternative), providing all data stored and used in the pixel shader during the lighting passes is in the same space.

World space pixel positions can be calculated by transforming each vertex in the vertex shader by the world matrix for the associated model and outputting the result to a 3D texture coordinate. The iterated texture coordinates in the pixel shader define the world space position for this pixel.

A 16-bit per-channel float format is ideal to store the pixel's world position in order to accommodate a wide range of possible values with sufficient precision (i.e., `D3DFMT_A16B16G16R16F`). See Color Plate 11 for an illustration of position data.

Pixel Normal Vector

This is the world space normalized normal vector for the pixel. There are two options that we can choose when producing the world space normal vectors for storage in the property buffer: model space and tangent space.

Pixel Normals Defined in Model Space

The first and simplest option is to have the normals in bump maps already defined in *model space*. Model space normal maps are sometimes calculated with a software tool to generate a low-poly model with model space normal maps from a high-poly model. With normals defined in model space, a simple world matrix transformation in the pixel shader is all that is required to obtain world space normals. Although this solution is simple to implement in terms of calculations, it also requires more texture memory, since texture maps are usually unique for each model. Also, the artist may have the scene's normal data already defined in *tangent space*, so it might be more convenient from a design point of view to use these instead.

Pixel Normals Defined in Tangent Space

The second option is more complicated but operates from normals defined in *tangent space* (also called *texture space*). Tangent space is a coordinate system indicating the texture surface orientation at each vertex. Traditional per-pixel lighting usually involves transforming the light vector into tangent space so that a DOT3 operation can be performed with the tangent space normals contained in the bump map. See Equation (1).

$$\vec{V}_{TS} = \vec{V}_{MS} \times (TS) = \vec{V}_{MS} \times \begin{pmatrix} T_X & N_X & B_X \\ T_Y & N_Y & B_Y \\ T_Z & N_Z & B_Z \end{pmatrix} \tag{1}$$

With deferred shading, the *normals* need to be transformed from tangent space to world space. In the same way a vector is transformed from model space to tangent space using the tangent space matrix, a tangent space vector can be transformed to model space using the *inverse* of the tangent space matrix. Conveniently, the tangent space matrix is a pure rotation matrix, so its inverse is simply its transpose. See Equation (2).

$$\vec{V}_{TS} = \vec{V}_{TS} \times \begin{pmatrix} T_X & N_X & B_X \\ T_Y & N_Y & B_Y \\ T_Z & N_Z & B_Z \end{pmatrix}^T = \vec{V}_{TS} \times \begin{pmatrix} T_X & T_Y & T_Z \\ N_X & N_Y & N_Z \\ B_X & B_Y & B_Z \end{pmatrix} \tag{2}$$

Because we need the normal vectors in world space, we also need to transform them with the rotation part of the world matrix associated with the current model. The equation becomes:

$$\vec{V}_{WS} = \vec{V}_{TS} \times (TS)^T \times (W) \tag{3}$$

For static models, it is a good idea to have their local orientation match their orientation in world space so that only a simple transformation with the transpose of the tangent space matrix is necessary. This saves a few instructions compared to the dynamic object's case, which requires Equation (3) to be fully honored.

Because we need a set of transposed tangent space vectors at each *pixel*, the normal, binormal, and tangent vectors are passed to the pixel shader through a

set of three 3D texture coordinates. The iterated vectors define our tangent space matrix for the current pixel. In theory, these vectors need renormalization before they can be used to transform the normal; however, in practice the difference in visual quality is negligible, provided the scene tessellation is high enough.

Precision and Storage

In both cases it can be desirable to renormalize the iterated normals for improved accuracy. Although linearly interpolated normals are usually close enough to unit vectors, the error margin accumulates when complex math operations are performed on these vectors (e.g., calculation of reflection vectors).

Although a float format could be used to store world space normal vectors in the attribute buffer, it is more economical and usually sufficiently accurate to use a 32-bit integer format instead (`D3DFMT_A2W10V10U10`, `D3DFMT_A2B10G10R10`, `D3DFMT_Q8W8V8U8`, `D3DFMT_A8R8G8B8`).

The deferred shading implementation described in this article uses tangent space normal maps. See Color Plate 12 for an illustration of normal data.

Pixel Diffuse Color

The pixel's diffuse color is stored in the property buffer. This color is extracted from the diffuse texture associated with the model. This color can be stored using a simple 32-bit texture format (`D3DFMT_A8R8G8B8`). Diffuse data is shown in Color Plate 13.

Building Pass

This section details how the attribute buffer is constructed and stored in MRTs. During the building pass, all the data relevant to the scene is calculated and stored in our MRTs. The pixel shader sends the relevant data into each of the three MRTs (i.e., pixel position, normal, and color).

Vertex Shader Code

The vertex shader code used for the building pass is fairly simple.

```
;-----------------------------------------------------------------------
; Constants specified by the app
; c0-c3 = Global transformation matrix (World*View*Projection)
; c4-c7 = World transformation matrix
;
; Vertex components
; v0         = Vertex Position
; v1, v2, v3 = Inverse of tangent space vectors
; v4         = 2D texture coordinates (model coordinates)
;-----------------------------------------------------------------------
vs_2_0

dcl_position v0          ; Vertex position
```

```
dcl_binormal v1          ; Transposed binormal
dcl_tangent  v2          ; Transposed tangent
dcl_normal   v3          ; Transposed normal
dcl_texcoord v4          ; Texture coordinates for diffuse and normal map

; Vertex transformation
m4x4 oPos, v0, c0        ; Transform vertices by WVP matrix

; Model texture coordinates
mov oT0.xy, v4.xy        ; Simply copy texture coordinates

; World space coordinates
m4x3 oT1.xyz, v0, c4     ; Transform vertices by world matrix (no w
                         ; needed)

; Inverse (transpose) of tangent space vectors
mov oT2.xyz, v1
mov oT3.xyz, v2
mov oT4.xyz, v3          ; Pass in transposed tangent space vectors
```

Pixel Shader Code

The pixel shader version used has to be able to output multiple color values. Therefore, pixel shader 2.0 or higher is required. The model texture coordinates are used to sample the pixel diffuse color and normal from their respective textures. The world space coordinates are directly stored into the position MRT. Finally, the transposed tangent space vectors are used to transform the sampled normal before storing it into the normal MRT.

```
;-----------------------------------------------------------------------
; Constants specified by the app
; c0-c3 = World transformation matrix for model
;-----------------------------------------------------------------------
ps_2_0

; Samplers
dcl_2d s0    ; Diffuse map
dcl_2d s1    ; Normal map

; Texture coordinates
dcl t0.xy    ; Texture coordinates for diffuse and normal map
dcl t1.xyz   ; World-space position
dcl t2.xyz   ; Binormal
dcl t3.xyz   ; Tangent
dcl t4.xyz   ; Normal (Transposed tangent space vectors)

; Constants
def c30, 1.0, 2.0, 0.0, 0.0
def c31, 0.2, 0.5, 1.0, 1.0

; Texture sampling
texld r2, t0, s1     ; r2 = Normal vector from normal map
```

```
texld r3, t0, s0     ; r3 = Color from diffuse map

; Store world-space coordinates into MRT#0
mov oC0, t1          ; Store pixel position in MRT#0

; Convert normal to signed vector
mad r2, r2, c30.g, -c30.r    ; r2 = 2*(r2 - 0.5)

; Transform normal vector from tangent space to model space
dp3 r4.x, r2, t2
dp3 r4.y, r2, t3
dp3 r4.z, r2, t4     ; r4.xyz = model space normal

; Transform model space normal vector to world space. Note that only
; the rotation part of the world matrix is needed.
; This step is not required for static models if their
; original model space orientation matches their orientation
; in world space. This would save 3 instructions.
m4x3 r1.xyz, r4, c0

; Convert normal vector to fixed point
; This is not required if the destination MRT is float or signed
mad r1, r1, c31.g, c31.g     ; r1 = 0.5*(r1 + 0.5)

; Store world-space normal into MRT#1
mov oC1, r1

; Store diffuse color into MRT#2
mov oC2, r3
```

We've already established that all models rendered with an identity rotation in their world matrix do not need to have their normals further transformed by this matrix. As shown above, skipping this step would save three pixel shader instructions. However, this implies using two different shaders, one for dynamic models and another for static ones. As these are usually not rendered in a defined order, the amount of swapping between the two pixel shaders could be excessive. Using extended pixel shader 2.0 (ps_2_x) or pixel shader 3.0, static flow control can be used to determine if transformation with the world matrix is needed.

Note that all texture filtering, such as trilinear or anisotropic, need only be performed in the building pass; the shading passes will directly access the already-filtered pixels from the MRT surfaces that we have rendered (using point sampling).

NOTE It is possible to store extra components into the attribute buffer using some form of compression. For instance, bias and scale operations would allow two 16-bit integer components to be stored in a 32-bit component. Pixel shader instructions are required to pack and unpack the data, but it enables more elements to be stored in case there are not enough outputs available.

Shading Passes

Each shading pass needs only to send a screen-aligned quad so that the shading calculations affect the entire render surface. The quad's texture coordinates have to be set up so that all pixels in the quad reference our MRT data at a 1:1 mapping ratio. Direct3D's sampling rules stipulate that an offset is required in order to achieve a perfect texel-to-pixel mapping. Given the width (W) and height (H) of the back buffer, the vertices forming the full-screen quad need to be set up in the following way:

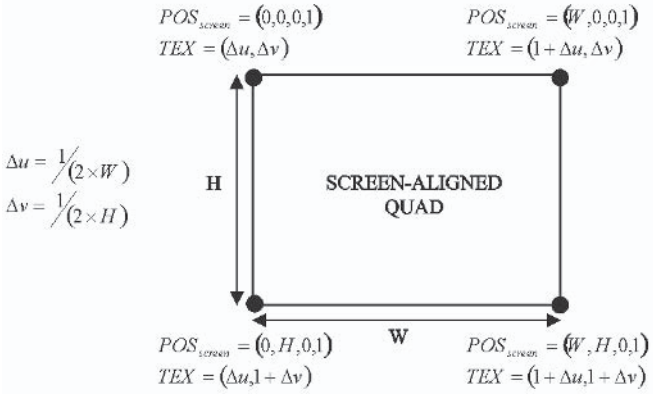

Figure 2: Setting up the vertex structure for a screen-aligned quad

POS_{screen} is the vertex position in screen space coordinates. *TEX* represents the 2D texture coordinates to use for this vertex.

NOTE The same 1:1 mapping ratio is achieved by offsetting screen space positions by –0.5 and setting texture coordinates to (0,0), (1, 0), (0, 1), and (1,1).

Because the contents of the MRTs already represent visible pixels, there is no need to enable depth buffering during the shading passes, saving valuable memory bandwidth.

Vertex Shader Code

No vertex processing is required, since screen space coordinates are directly sent to the pixel shader. To skip vertex processing, the vertex declaration must include the `D3DDECLUSAGE_POSITIONT` definition. In this case, only the transformed position and the texture coordinates are required, thus the vertex declaration is:

```
D3DVERTEXELEMENT9 declTPositionUV[] =
{
 { 0, 0, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITIONT, 0},
 { 0, 16, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
```

```
 D3DDECL_END()
};
```

Pixel Shader Code

The calculations to be performed during the shading passes obviously depend on the needs of the application. It is up to the programmer to decide what lighting or special effects can be applied with the data available in the attribute buffer. The actual shading algorithms are not affected by their transition to a deferred shading context.

The following pixel shader is based on Phong lighting and implements diffuse and specular shading with distance attenuation. Each active light's contribution is accumulated by additively blending a full-screen quad onto the frame buffer. A number of optimizations can be implemented in this shader, but for simplicity they are not shown here. For example, another lookup texture could be used to replace the 3-slot `pow` instruction or a simpler falloff model could be used, etc. A cube normalization map is used for vector normalization, although the `nrm` instruction is an alternative choice. The falloff texture is a simple lookup texture containing the light attenuation based on the pixel's distance from the light divided by its maximum range (texture clamping is used).

```
;------------------------------------------------------------------------
; Constants specified by the app
; c0 : light position in world space
; c8 : camera position in world space
; c22: c22.a = 1/(light max range), c22.rgb = 1.0f
;------------------------------------------------------------------------
ps_2_0

; Samplers
dcl_2d s0    ; MRT#0 = Pixel position in world space
dcl_2d s1    ; MRT#1 = Pixel normal vector
dcl_2d s2    ; MRT#2 = Pixel diffuse color
dcl_2d s3    ; Falloff texture
dcl_cube s4  ; Cube normalization texture map

; Texture coordinates
dcl t0.xy                    ; Quad screen-space texture coordinates

; Constants
def c20, 0.5, 2.0, -2.0, 1.0
def c21, 8.0, -0.75, 4.0, 0.0

; Retrieve property buffer data from MRT textures
texld r0, t0, s0             ; r0.xyz = Pixel world space position
texld r2, t0, s1             ; r2.xyz = Pixel normal vector
texld r3, t0, s2             ; r3.rgb = Pixel color

; Convert normal to signed vector
; This is not required if the normal vector was stored in a signed
```

```
; or float format
mad r2, r2, c20.y, -c20.w    ; r2 = 2*(r2 - 1)

; Calculate pixel-to-light vector
sub r1.xyz, c0, r0           ; r1 = Lpos - Vpos
mov r1.w, c20.w              ; Set r1.w to 1.0
nrm r4, r1                   ; Normalize vector (r4.w = 1.0/distance)

; Compute diffuse intensity
dp3 r5.w, r4, r2             ; r5.w = (N.L)

; FallOff
rcp r6, r4.w                 ; r6 = 1/(1/distance) = distance
mul r6, r6, c22.a            ; Divide by light max range
texld r6, r6, s3             ; Sample falloff texture

; Compute halfway vector
sub r1.xyz, c8, r0           ; Compute view vector V (pixel to camera)

texld r1, r1, s4             ; Normalized vector with cube map
mad r1, r1, c20.y, -c20.w    ; Convert vector to signed format

add r1, r1, r4               ; Add view and light vector
texld r1, r1, s4             ; Normalize half angle vector with cube map
mad r1, r1, c20.y, -c20.w    ; Convert to signed format

; Compute specular intensity
dp3_sat r1.w, r1, r2         ; r1.w = sat(H.N)
pow r1.w, r1.w, c21.r        ; r1.w = (H.N)^8

; Set specular to 0 if pixel normal is not facing the light
cmp r1.w, r5.w, r1.w, c21.w  ; r1.w = ( (N.L)>=0 ) ? (H.N)^8 : 0

; Output final color
mad r0, r3, r5.w, r1.w       ; Modulate diffuse color and diffuse
                             ; intensity and add specular
mul r0, r0, r6               ; Modulate with falloff
mov oC0, r0                  ; Output final color
```

Advanced Shading Passes

The simple lighting pass shown above is only the beginning of what can be achieved with deferred shading. Because shading passes are no longer geometry-dependent and only applied to visible geometry, their utilization is optimal. This performance saving can be used to implement more complex shaders or effects.

Better Lighting

A "real" specular calculation based on the camera reflection vector is straightforward to implement, since both the pixel and camera position are known. That is, instead of calculating $(\vec{H}.\vec{N})$, the light reflection vector around the normal (given by $\vec{R} = 2 \times (\vec{N}.\vec{L}) \times \vec{N} - \vec{L}$) can be calculated and used in a dot product operation with the view vector. The specular properties of each pixel could be stored in the attribute buffer (e.g., using the alpha of the diffuse texture in our implementation) and used as the power of the specular calculation.

Different types of light can be implemented, from directional and point lights to spotlights or custom-shaped lights (light volumes are discussed later in this article). More complex light attenuation models can be implemented using math instructions inside the pixel shader instead of a texture lookup. Light maps can be used for custom-shaped lights or when there is a need to restrict light contribution to a defined area. The demo on the CD offers some implementations of the effects mentioned above.

Extended Attribute Buffer

Providing memory storage and bandwidth are not limiting factors (see later in this article for a discussion about bandwidth considerations), the attribute buffer can be used to store additional data relevant to the scene properties, allowing more complex effects to be implemented later on in the shading passes.

In the example shown in this article, the material properties are simply approximated to a diffuse color issued from each model's diffuse map. Other properties, like specular maps, specular power maps, detail maps, tangent vectors for anisotropic lighting, a Fresnel term, BRDF data, etc., could be stored in the attribute buffer. There are potentially so many material properties to store that there might not be enough space to accommodate them. Various tricks can be used to effectively compress as much data as possible in a reasonably allocated MRT space, like storing material IDs, using volume textures, etc.

Shadows

Deferred shading is fully compatible with the rendering of shadows within the scene.

Let's consider stencil shadows (as used in the demo). In the traditional rendering case (e.g., Doom III-style rendering), each shadow-casting light requires the scene geometry to be submitted (on top of shadow volumes) so that only non-shadowed areas are affected by the lighting calculations. When a high number of lights are used, the number of geometry passes quickly becomes overwhelming. With deferred shading, only the shadow volumes and a full-screen quad per light are required:

```
Building pass
For each light:
    Clear stencil buffer, disable color writes
```

Render shadow volumes onto stencil buffer
Enable color writes, stencil test passes for non-shadowed areas
Light pass

Shadow maps are also compatible with deferred shading. Since the world space pixel position can be stored in the attribute buffer, the distance from the pixel to the current light can be calculated in the shading pass. It can then be compared to the depth value in the shadow map to determine if the pixel is in shadow or not. This technique is sometimes referred to as forward shadow mapping.

Higher Color Range

DirectX 9 does not allow blending into floating-point render targets. Because of this limitation, alpha blending cannot be used to accumulate high dynamic range lighting calculation results into a destination frame buffer in float format. With fixed-point render targets, color channels are automatically clamped to 1.0 during the blending process, and so if a higher color range is not needed, it is more economical and straightforward to blend all shading contributions into a fixed-point frame buffer. High color range effects, however, require an alternative solution.

The most obvious workaround to this limitation is to ignore the hardware blending capabilities and perform alpha blending on the pixel shader manually. The idea is to alternate between two floating-point render targets; one contains the current contents of all shading contributions prior to the current pass and is set as a texture input to the pixel shader, while the other is set as the destination render target and receives the new color values updated with the contributions of the current pass.

Translucency

In traditional rendering, alpha-blended polygons require a rendering pass of their own. With deferred shading, alpha polygons requiring shading (e.g., stained-glass objects) also need separate processing, although punch-through polygons (i.e., alpha-tested) can be sent with all other opaque triangles during the building pass. This is because the attribute buffer can only store properties for a single pixel; thus, any translucent pixels rendered on top of these would need to be stored elsewhere. Unless additive blending is used, real translucent pixels (using an order-dependent blending mode like SRCALPHA-INVSRCALPHA) need to be blended with the back buffer only *once* to avoid repeated contributions from themselves. Hence, all shading passes need to be performed on the translucent pixels *before* any blending with the background takes place. Another problem is overlapping translucency; any translucent pixel rendered on top of a previously rendered translucent pixel will need to take into account the new pixel color at this location (i.e., resulting from the previous blending). These two points are of paramount importance, since ignoring them can cause visual errors in the resulting render.

Unfortunately, there is no easy way to overcome these problems. For a real-time application relying on performance, the best idea is probably to avoid

using translucent objects that require lighting (light-independent translucent objects like explosions, laser beams, smoke, etc., are unaffected because they typically do not need to be shaded). One could attempt to concatenate all shading passes in one large shader and gross-sort the translucent triangles from back to front using this shader. However, this might not be practical because of the sampler and instruction limits in the pixel shader. Furthermore, this might not be compatible with some shadowing techniques like stencil buffering. If lit translucent objects *are* a requirement of your scene and you are prepared to "go all the way," then a solution is *depth peeling* (i.e., shading each "layer" of translucency separately before blending it with the frame buffer). In practice the additional memory, performance, and complexity caused by depth peeling do not make it a very attractive solution to the translucency problem inherent in deferred shading.

Deferred Shading Optimizations

MRT Configuration and Bandwidth

Care must be taken when choosing the format and bit depth of the MRTs to use with deferred shading. With DirectX 9 support for IEEE float formats, one might be tempted to use four MRTs in the `D3DFMT_A32B32G32R32F` format for the added precision and robustness they bring. However, not only will the memory requirements be considerable (at a 1024x768 screen resolution, this corresponds to total of 48MB), but also the required memory bandwidth for each frame is likely to cause a performance bottleneck.

Let's examine in detail how much memory bandwidth deferred shading requires. We define the following variables and assume some practical values for them:

Variable Name	Description	Values	Justification
W, H	Render width, height	1024, 768	Typical rendering resolution
Z_{BPP}, BB_{BPP}	Depth/stencil buffer bit depth, back buffer bit depth	32 bpp, 32 bpp	Typical depth/back buffer bit depth
Overdraw	Average overdraw per pixel	3	Average overdraw
Sorting	Average number of pixels arranged in a back to front order. In the worst case where all pixels are ordered back to front this value equals *Overdraw*; in the best case it will be equal to 1: $1 \leq Sorting \leq Overdraw$.	1.5	We assume that half of overdrawn pixels are drawn back to front.
T_{BPP}	Average texture bit depth (e.g., if half of textures are 32bpp and other half is 16bpp, then $T_{BPP}=24$)	24 bpp	Mix of compressed, 32bpp and 32bpp+ textures

Variable Name	Description	Values	Justification
T_B, T_S	Average number of texture lookups in building pass/shading pass pixel shader	2, 4	Two texture lookup in building pass (e.g., diffuse and normal map), four in shading pass (e.g., light maps, cube normalization map, etc.)
n	Number of shading passes to perform	8	Average of eight full-screen shading passes
n_{MRT}	Number of MRT surfaces: $1 \le n_{MRT} \le 4$	-	Variable
MRT_{BPP}	MRT bit depth	-	Variable

Table 1: Description of variables used in bandwidth calculation

Assumptions

Because of different 3D acceleration implementations among graphic adapters, it can be difficult to model a universal bandwidth equation. However, by making a number of assumptions about the rendering environment, we can get close to an accurate result. Firstly, we assume the target 3D accelerator is an immediate mode renderer with some form of early Z support (i.e., the depth test is performed before the pixel shader — we can reasonably expect this optimization from all new DX9 accelerators). Secondly, we assume a texture sample needs only a single texel fetch, regardless of texture filtering. Finally, we ignore any form of cache effectiveness.

Let's see how much memory needs to be transferred across the bus for a single frame using deferred shading. We start by analyzing each feature requiring memory traffic during the building pass:

Depth/Stencil: $W \times H \times Z_{BPP} \times (Overdraw + Sorting)$

Back Buffer: 0 (the back buffer is not read nor written during the building pass)

Textures: $W \times H \times T_{BPP} \times T_B \times Sorting$

Geometry buffers (vertex and index buffers): $C_{Geometry}$ (constant value)

MRTs: $n_{MRT} \times W \times H \times MRT_{BPP} \times Sorting$

Adding these together, we get:

$$Memory_{OneFrame} = W \times H \times [Z_{BPP} \times (Overdraw + Sorting) + T_{BPP} \times T_B \times Sorting + n_{MRT} \times MRT_{BPP} \times Sorting] + C_{Geometry} \quad (4)$$

Let's now examine how much memory needs to be transferred across the bus during the n shading passes.

Depth/Stencil: 0 (depth buffer disabled)

Back Buffer: $2_{(R/W)} \times W \times H \times BB_{BPP} \times n$

Textures: $W \times H \times T_S \times T_{BPP} \times n$

Geometry buffers: 0

MRTs: $n_{MRT} \times W \times H \times MRT_{BPP} \times n$

Adding these together we get:

$$Memory_{OneFrame} = W \times H \times n \times \lfloor 2_{(R/W)} \times BB_{BPP} + T_S \times T_{BPP} + n_{MRT} \times MRT_{BPP} \rfloor \quad (5)$$

By adding the amounts of external memory accesses to perform during the building and the n shading passes and multiplying by a desired frame rate of 60 fps, we obtain the following bandwidth formula:

$$Bandwidth_{60\,fps} = \left(W \times H \times \begin{bmatrix} MRT_{BPP} \times n_{MRT} \times (Sorting + n) \\ +Z_{BPP} \times (Overdraw + Sorting) \\ +T_{BPP} \times T_B \times Sorting \\ +n \times (2 \times BB_{BPP} + T_S \times T_{BPP}) \end{bmatrix} + C_{Geometry} \right) \times 60\text{Bytes / Sec} \quad (6)$$

Practical Example

Using our practical values, the bandwidth equation becomes:

$$Bandwidth_{60\,fps} = \left(1024 \times 768 \times \begin{bmatrix} MRT_{BPP} \times n_{MRT} \times (1.5 + 8) \\ +8 \times (3 + 1.5) \\ +3 \times 2 \times 1.5 \\ +8 \times (2 \times 4 + 4 \times 3) \end{bmatrix} + 16.10^6 \right) \times 60\text{Bytes / Sec}$$

$$Bandwidth_{60\,fps} = 0.05 \times [MRT_{BPP} \times n_{MRT} \times 9.5 + 205] + 1\text{GBytes / Sec}$$

A gross approximation of the end result is:

$$Bandwidth_{60\,fps} = \frac{n_{MRT} \times MRT_{BPP}}{2} + 10\text{GBytes / Sec} \quad (7)$$

Here are some examples of bandwidth values for a defined number of MRTs and their bit depth for the selected variables:

	MRT_{BPP}=32 bpp	MRT_{BPP}=64 bpp	MRT_{BPP}=128 bpp
n_{MRT}=2	**14** GBytes/Sec	**18** GBytes/Sec	**26** GBytes/Sec
n_{MRT}=3	**16** GBytes/Sec	**22** GBytes/Sec	**34** GBytes/Sec
n_{MRT}=4	**18** GBytes/Sec	**26** GBytes/Sec	**42** GBytes/Sec

Table 2: Bandwidth figures for various MRT configurations with selected variables

This table clearly shows the overwhelming bandwidth requirements when an unreasonable MRT configuration is chosen. Of course, other factors influence the bandwidth requirements (notably the number of shading passes to perform), but overall it is wiser to select the minimum possible number and bit depth for MRTs, which can accommodate the stored data.

Optimized MRT Configuration

The implementation described in this article stores position, normal, and diffuse color into MRTs, totaling 64+32+32=128 bits of data. Providing the 3D device supports the `D3DPMISCCAPS_MRTINDEPENDENTBITDEPTHS` cap, this data can be rearranged in an optimized configuration so that memory bandwidth and footprint are reduced. Consider the following configuration of four MRTs:

- MRT#0: `D3DFMT_G16R16F`: Store X, Y position in Red and Green channels
- MRT#1: `D3DFMT_R16F`: Store Z position in Red channel
- MRT#2: `D3DFMT_A8R8G8B8`: Store diffuse color in RGB, normal Z in A
- MRT#3: `D3DFMT_A8L8`: Store normal X in A, Y in L

This equates to a total of 96 bits. Note that the 3D device has to support these formats as render targets for this configuration to be valid. The pixel shader code in the building and shading passes needs to be adjusted to write and read MRT data into their appropriate components.

Using 2D Shapes to Optimize Shading Passes

Sending full-screen-aligned quads for all shading passes results in all the screen pixels being processed by the pixel shader. Depending on the number of passes to perform, screen resolution, and shader complexity, this can lead to excessive and costly pixel processing. By knowing the boundaries at which a light or special effect ceases to contribute to the scene, only the screen pixels that are inside those limits need to be sent (e.g., for a point light, this would be a 2D-projected sphere whose radius is the maximum range of the light). Explosions, cone lights, etc., can also benefit from using 2D shapes during the shading passes.

Projected Volumes

Unless the screen coordinates to which an effect is to be applied are already known (HUD, part-screen filters, etc.), in most cases the 2D shapes to deal with will be projections of 3D volumes into screen coordinates. This is required so that world space units can be correctly transformed and mapped into screen space.

Let's consider a simple point light of maximum range, MaxRange. Instead of sending a full-screen quad during the shading pass, a sphere of radius MaxRange is transformed and rasterized. This results in only the pixels inside the sphere to be affected by the light and thus processed. Note that the pixel shader's falloff calculation must match the range used by the volume (i.e., pixels whose distance from the light source is greater than MaxRange have a falloff of zero); otherwise the difference in intensity between pixels inside and outside the sphere will be clearly visible.

Sending a volume will have the same visual result as sending a full-screen quad but without the overhead of processing out-of-bounds pixels. Color Plate 14 demonstrates this idea of processing only those pixels that are affected by the light.

Back Faces Only

Because the camera could be located *inside* the volume, it is important to render the *back faces* of the volume only. Providing the volume is closed, is convex, and does not intercept the far clip plane, this will ensure the projected shape is always visible on the screen (for non-convex light volumes, a convex bounding volume should be used). If the culling order was not reversed for those volumes, then the projected shapes would only be correct if the camera was outside the volumes. Rendering back faces only ensures the projected shape is correct regardless of the camera position. Note that turning back-face culling off completely is not a solution because some screen pixels end up being processed twice.

Because we are interested in the intersection of the light area with the visible pixels in the scene, a further optimization to this technique is to only render the back-facing pixels of a volume whenever the normal depth buffer visibility test fails. This can be achieved by simply inverting the Z test when rendering the volume (e.g., using `D3DCMP_GREATER` instead of `D3DCMP_LESSEQUAL`).

Mapping Volume Texture Coordinates to Screen Space

Calculating screen space texture coordinates for the 2D projection of a volume is more complicated than for an already-transformed full-screen quad. Given the homogenous clipping coordinates x_H, y_H, z_H, w_H and the back buffer dimensions Width and Height, the equation to retrieve screen-space texture coordinates u_S and v_S is given by:

$$\begin{aligned} u_S &= \frac{1}{2} \times \left(\frac{x_H}{w_H} + 1 \right) + \frac{1}{2 \times Width} \\ v_S &= \frac{1}{2} \times \left(1 - \frac{y_H}{w_H} \right) + \frac{1}{2 \times Height} \end{aligned} \tag{8}$$

Although the vertex shader could pass homogenous clipping coordinates directly to the pixel shader, a sensible optimization is to precalculate the texture coordinates in the vertex shader so that the pixel shader need only perform the projective divide. Conveniently, the latter can be obtained during the sampling process by projected texture lookup. The equation becomes:

$$\begin{aligned} u_S &= \frac{1}{w_H} \times \left(\frac{x_H}{2} + \frac{w_H}{2} + \frac{w_H}{2 \times Width} \right) \\ v_S &= \frac{1}{w_H} \times \left(\frac{w_H}{2} + \frac{y_H}{2} + \frac{w_H}{2 \times Height} \right) \end{aligned} \tag{9}$$

Vertex Shader Code

```
;------------------------------------------------------------------------
; Constants specified by the app
; c0-c3 = Global transformation matrix (World*View*Projection)
; c9    = 1.0f/(2.0f*dwScreenWidth), 1.0f/(2.0f*dwScreenHeight)
```

```
;
; Vertex components
; v0    = Vertex Position
;------------------------------------------------------------------
vs_2_0

dcl_position v0                 ; Vertex position

def c8, 0.5, -0.5, 0.0, 0.0

; Vertex transformation
m4x4 r0, v0, c0                 ; Transform vertices by WVP matrix
mov oPos, r0                    ; Output position

; Compute texture coordinates
mul r0.xy, r0, c8               ; x/2, -y/2
mad r0.xy, r0.w, c8.x, r0       ; x/2 + w/2, -y/2 + w/2
mad r0.xy, r0.w, c9, r0         ;  x/2 + w/2 + w/(2*Width),
                                ; -y/2 + w/2 + w/(2*Height)

mov oT0, r0                     ; Output texture coordinates
```

Pixel Shader Code

The iterated texture coordinates issued from the vertex shader are divided by w_H and used as texture coordinates to sample the MRT textures using a projected texture lookup.

```
ps_2_0

; Texture coordinates
dcl t0.xyzw                 ; iterated texture coordinates

; Samplers
dcl_2d s0                   ; MRT#0 = World space position
dcl_2d s1                   ; MRT#2 = World space normal vector
dcl_2d s2                   ; MRT#3 = Pixel Diffuse Color

; Projected texture lookup into MRT textures
texldp r0, t0, s0           ; r0 = world space position
texldp r1, t0, s1           ; r1 = world space normal
texldp r2, t0, s2           ; r2 = Diffuse map color
```

NOTE With pixel shader 3.0 support, the process of retrieving texture coordinates for a projected shape is simpler; the position register (which contains the x, y screen position of the pixel currently being processed) can be used with some scaling and biasing to perform this calculation directly.

Using shapes is better suited for smaller lights and special effects with limited range. For bright lights or global effects likely to affect the entire render, a full-screen quad is a better solution.

CD Demo

The demo on the companion CD shows a scene using deferred shading (see Color Plate 15). A total of nine lights are used — one point light affecting the entire screen ("full-screen" light), two point lights rendered using projected sphere shapes, and six cone lights rendered using projected cone shapes. A DX9-class 3D accelerator with vertex shader 2.0/pixel shader 2.0 and MRT support is required to run this demo.

The application can be controlled using menu options (press Alt to bring up the menu in full-screen mode). The latest version of the demo is available on the PowerVR Developer Relations web site at www.pvrdev.com.

Summary

This article described deferred shading and showed how to implement this technique using multiple render targets in DirectX 9. The two-phase process of the algorithm was detailed with shader code examples as supportive material. Some advanced deferred shading effects were proposed, and the robustness and overall simplicity of the technique should encourage graphics programmers to invent their own ideas for even better usage of this rendering algorithm. Finally, performance and memory footprint considerations were discussed, and optimizations were suggested to improve the effectiveness of the technique.