2.5

Overcoming Deferred Shading Drawbacks

Frank Puig Placeres
(*fpuig@fpuig.cjb.net*)

Lighting in today's applications is performed by batching light sources together into small groups of three to eight lights that can be managed by current shaders. For each of these groups, the scene is rendered and the light's contribution is added into the frame buffer that, at the end, contains the influence of all lights in the world.

When targeting today's graphics, it may be justified to use a multipass solution for lighting, but that can only be justified when targeting graphics that generally consist of low- and medium-poly-count scenes with no complex materials, a very small number of light types, and where illumination comes from a few lights spread all over the scene.

However, next-generation games have heavily increased the poly count and number of lights on screen while raising the material's complexity. Performing several passes on those heavy scenes clearly overwhelms the hardware capabilities even when most of the passes can be accelerated by the use of early rejection features.

Special effects such as high dynamic range, depth of field, heat haze, and dynamic volumetric fogs, among others, are going to be the standard; this means that traditional multipass rendering systems are next to impossible because most of those effects require rerendering the scene several times.

A better solution is presented that uses deferred shading. This technique overcomes most of the above drawbacks while simplifying the rendering of multiple special effects. It also reduces the overhead of performing several passes on the scene by reducing it to only rendering a full screen quad for each pass instead of the complete scene geometry.

Deferred shading also allows the reduction of lighting, shadow, special effects, complex materials, and other computations on the pixels that are visible, which is required when materials get really heavy and lighting involves more realistic models. In addition, forward shadow mapping fits incredibly easily on a deferred shading system. Not only do shadows work well with the system, but so do other special effects since everything becomes image postprocessing work.

Nonetheless, deferred shading has several disadvantages. It needs a lot of memory to store the geometric buffer, it produces a noticeable impact on fill rate, it can't handle transparency efficiently, and it can suffer from antialiasing effects.

This article presents solutions to overcome or reduce most of the deferred shading drawbacks and gives tips on how to implement a scalable system that can run on more graphic cards. The system uses less memory and optimizes the whole process not only in the shaders but also from the application itself by implementing high-level managers that run on the CPU.

Memory Optimization

Contrary to classic forward rendering, in a deferred shading system, lighting and other special effects are not computed in the same pass in which the scene geometry is processed. Instead, there is a first pass in which the scene geometry is rendered and per-pixel attributes such as `Position` and `Normal` are saved into several textures composing an auxiliary buffer called a *geometric buffer* (G-buffer). Those textures are then used on subsequent passes to get the pixel's geometric data without processing the scene again.

As there is no need to reprocess the scene's geometry in each pass, executing multiple passes to create special effects becomes simple 2D image postprocessing. However, using the G-buffer comes with its own drawbacks. For each pixel, all the geometrical attributes that are needed for lighting, shadow, or any other effect must be saved into the buffer's texels.

Common attributes such as `Position` and `Normal` can use up to three floats each, and there's also a need to store material values such as `specular power`, `glow factor`, and `occlusion term`, among others. This can increase the memory footprint between 10 and 40 MB just to store auxiliary values for standard game resolutions.

With the increase in the memory footprint comes the need to use multiple render targets or perform multiple passes to save all the values in the G-buffer. Next, we are going to present some tips to reduce the memory footprint and the other problems.

When designing a deferred shading system, special care must be taken with respect to which values should be stored in the G-buffer. Cleverly packing those values allows the reduction of the memory footprint.

Normal Vector

The easiest and a faster way to store the normal is to save each of its components to the G-buffer. However, knowing that each normal component is a float, it would take 12 bytes per pixel to store them.

Using normals that are unit length vectors, it's possible to compute one component, given the other two, by applying the equation:

$$z = \pm\sqrt{1 - x^2 - y^2}$$

The computed component could be a positive or negative number. However, if all the lighting is performed in view space, then the front-faced polygons are always going to have negative or positive Z components, depending on the frame of reference used. Thus, it is possible to only save the *x* and *y* component in the G-buffer and recreate the complete vector using the stored data.

The math involved in unpacking the *z* component can add up to five extra lines to the shader code. Those instructions can be safely removed by using a simple look-up texture, having in each texel the value of the *z* component corresponding to its *u* and *v* values. So a texture fetch where the *u* and *v* values correspond to the *x* and *y* normal components returns the *z* without extra math instructions.

Using the texture look-up could help improve performance on some applications and make things slower on others. This depends on the texture cache and how much texture bandwidth the application uses. So when replacing the math code with the texture fetch, it is advisable to compare the performance of both paths.

Further optimizations could reduce the precision in which the normal components are stored from a whole float down to a single byte or even just four to five bits per component. A byte is commonly used for a normal component when dealing with normal maps or other precomputed textures. It creates a nice trade-off between graphics quality and memory consumption.

Position

Instead of storing all the three components of the position vector, it's possible to only save one value in the G-buffer. Saving the distance the pixel is from the camera allows recreation of the complete position vector. When rendering the first pass (geometry phase), the position of the pixel in view space is stored as a single value into the buffer by using:

```
G_Buffer.z = length(Input.PosInViewSpace);
```

When it's necessary to retrieve the position from the G-buffer, the following code is used:

```
(Vertex Shader)
out.vEyeToScreen = float3( Input.ScreenPos.x * ViewAspect,
                   Input.ScreenPos.y, invTanHalfFOV );

(Pixel Shader)
float3 PixelPos = normalize(Input.vEyeToScreen) * G_Buffer.z;
```

This code computes a ray from the eye location to the screen position of the pixel, maintaining the view aspect that defines the relation between the screen width and height and the field of view angle. The ray is normalized and then multiplied by the distance from the eye location to the pixel, generating the position of the pixel as shown in Figure 2.5.1.

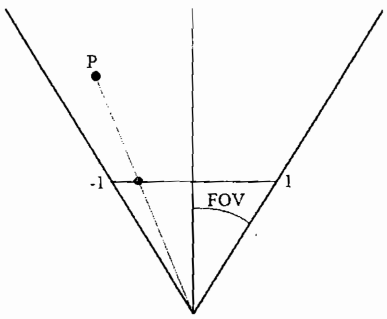

FIGURE 2.5.1 Generating the pixel position from the distance to the eye.

When storing the computed distance from the eye to the pixel position in the G-buffer, the value can be saved using several sizes, from a 32-bit float down to 24 or 16 bits. The selected size determines the final image quality, but as shown in Figure 2.5.2, these lower-precision results can be acceptable as fallback solutions because most of the time the results are indistinguishable from the high-resolution version.

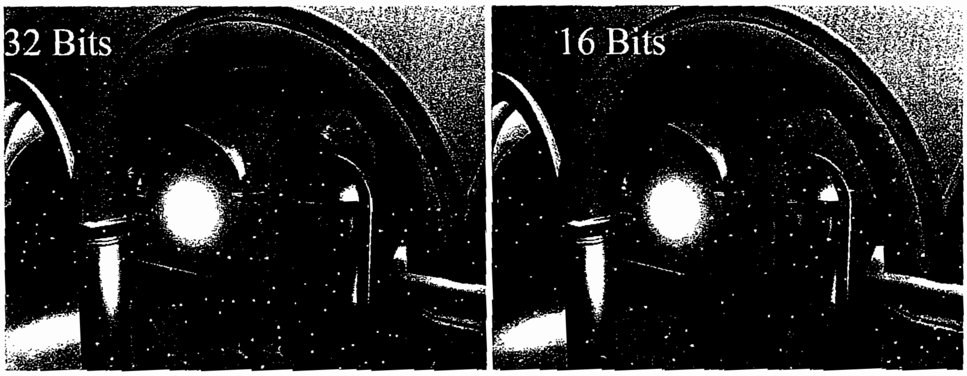

FIGURE 2.5.2 Scene rendered using different depth precisions.

Using 24 or 16 bits will not only match the depth buffer resolution and save memory space, but also gets rid of floating point textures on systems that don't support them. In later sections, we will present how it also allows better integration into the G-buffer, which will reduce the number of render targets used or the number of passes to be performed to fill the Geometry buffer.

Functions to pack and unpack a float 32 into 24 bits are presented in [Buttza05]. A similar approach can be followed for other target precisions or for changing the game view range.

Material Attributes

While it could be ideal to store all material attributes such as `specular power`, `glow factor`, `occlusion term`, and so on in the G-buffer, they consume too much memory. Applications that use one or two attributes could store them directly by first adjusting the number of bits assigned to them. For example, when the specular power is constrained to the values [1, 4, 10, 30], it can be saved using two bits, which gives four distinct values. The value then can be computed from the component inside the shader.

In most situations more attributes are needed; however, they don't change their value per pixel, but per surface. Where there are just a few materials in the scene, it's possible to pack all the attributes describing a material and only save an index of the material into the G-buffer. Depending on the number of materials in the scene, the index can be adjusted to use a certain number of bits that fits the maximum value.

Depending on the number of materials and the graphics hardware, the material attributes can be accessed in one of several ways. When all the attributes fit in the available unused shader constants, the shader is fed input values that can be used to do an indexed look-up.

When the number of materials and attributes exceeds the unused shader constants, the values can be packed into a texture where each row represents a different material, and the attributes can be accessed from the shader by using a texture fetch.

Designing the G-Buffer

Selecting the number of textures comprising the G-buffer, their format, and how the needed values are going to be distributed is one of the aspects that will deeply influence the resulting performance and memory footprint.

When designing the configuration of the G-buffer, special care must be taken to take advantage of the available features in the graphic hardware. Are multiple render targets supported? Can the hardware use floating point textures? What about other formats such as A16B16G16R16? How complex are the shaders that produce the lighting and special effects? Will the application rely heavily on texture access?

Several combinations appear to produce faster results, while others will reduce the memory footprint and minimize texture cache misses. It's possible to store values without packing them so they can be easily recovered in the lighting phase. However, this technique will consume a large amount of memory and will involve a higher number of passes to fill the G-buffer. It might also require multiple render targets and will rely on the target texture caches that might increase cache misses and texture fill rate.

ON THE CD

Other configurations can use lower-resolution targets by packing the elements before storing them. The examples on the CD-ROM show several combinations to pack the `Diffuse Color`, `Normal Vector`, `Position`, and `Material` into two RGBA8 textures with different precision ranges. Those configurations heavily reduce cache misses and the memory footprint but introduce some overhead while packing and unpacking the values of the G-buffer. Some of the other configurations that are possible use precision ranges that can diminish the graphic quality of the resulting image, but generally the quality is acceptable.

Fill-Rate Reduction

Given that deferred shading systems rely on filtering image pixels, they are likely to become fill-rate limited. Most implementations, after filling the G-buffer, just loop through every light source and special effect and apply the shader to each pixel. This is done by rendering a full-screen quad over the scene using the G-buffers as a source and letting the shader combine the values for the final output. When using this technique the shader will still have to process all view-port pixels, including those that are not affected by the light or the effect.

Instead of linearly looping through the list of lights and special effects and sending each of them to the deferred shading pipeline, it's possible to implement a high-level manager. The application acts as a firewall, only sending to the pipeline the sources that influence the final image and executing the shaders only on the pixels that are influenced by the effect or light.

The high-level manager receives the list of all sources that should modulate the resulting render and all the information about them. One such piece of information is a bounding object that contains information on how strong the source effect should be applied to the rendering equation (for a light it could be the brightness, etc.).

Using that list, the manager executes two phases. The first phase is called the social phase since it performs general algorithms to reduce the number of sources that have to be sent to the pipeline. The second phase is called the individual phase and will configure the shaders to reduce the processing cost of each source.

Social Stage

During this first stage, the manager filters the lights and effects on the scene, producing a smaller list of sources to be processed. To filter the list, the following pseudo code is applied:

1. Execute visibility and occlusion algorithms to discard lights whose influence is not appreciable.
2. Project visible sources bounding objects into screen space.
3. Combine similar sources that are too close in screen space or influence almost the same screen area.
4. Discard sources with a tiny contribution because of their projected bounding object being too small or too far.
5. Check that more than a predefined number of sources do not affect each screen region. Choose the biggest, strongest, and closer sources.

In the first step, using the source-bounding object, the manager finds which lights and effects are not occluded by the scene geometry and are inside the view frustum, thus affecting the visible screen. Sources that are not visible are discarded.

Subsequent steps analyze the source influence on the screen using the projection of the source-bounding object into screen space. As shown, there is no restriction about which bounding object can be used. It is possible to combine several objects to define the source region of influence. When projecting, all bounding objects are transformed into bidimensional shapes describing the source area of influence.

Further reductions to the source list produce a nonconservative result, which implies a loss of image quality. However, properly adjusting the settings of the next steps minimizes the quality impact. In most cases it is better to have a performance improvement than trying to achieve perfect images when the results are almost indistinguishable from the lower quality image.

Once the bounding objects have been projected into screen space, sources that are too close and affecting almost the same region are combined (see Figure 2.5.3a). These sources are combined into one source and then are applied using an intensity that simulates the result of the mixed sources.

Isolated sources that are too far or have an area of influence of just a few pixels can be omitted. Remember that the visual quality difference may not be worth the expensive operation of setting all the stages in the graphics pipeline to process these few pixels. An example is shown in Figure 2.5.3b.

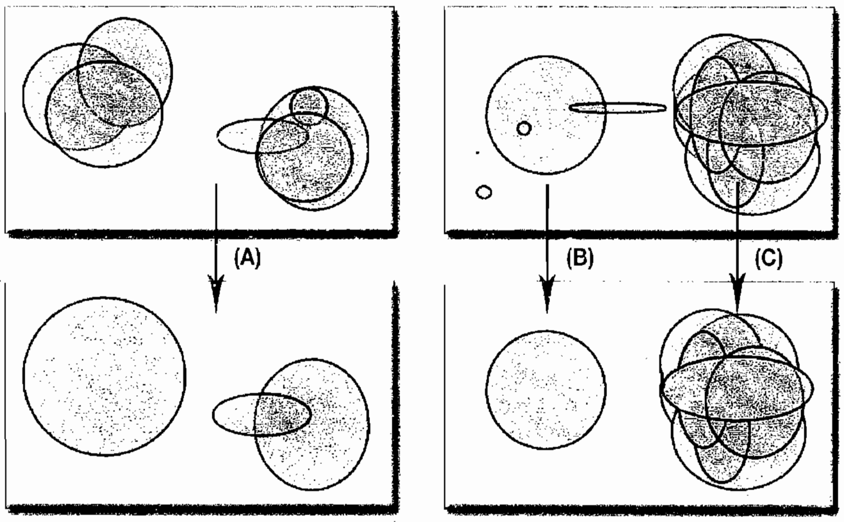

FIGURE 2.5.3 (A) Similar sources are combined. (**B**) Small sources are discarded. (C) There are no more than a fixed number of sources per pixel.

The previous actions are targeted to reduce the number of sources being processed according to their geometrical and visual properties. However, a last step can be added to help application performance. In this step the application can limit the number of sources affecting each pixel. For quicker computations, screen regions can be used instead of pixels. This is shown in Figure 2.5.3c.

When the average frame rate is high enough, the limiting number is raised. If the frame rate drops, the limiting number is lowered. This is a fast way to control the fill rate of the application and allows maintenance of smooth frame rates. A similar approach can be used for steps 3 and 4. Having variables track how far or small the sources should be before they are discarded or how different the sources should be, so that they are not combined, helps performance. These factors can also be tweaked according to the resulting performance.

Individual Stage

Lights and special effects in general can be classified in two main groups: global sources and local sources. Each group allows custom setup and optimization for the graphics pipeline to minimize overhead. The individual phase of the high-level manager classifies each source in the list returned by the social phase and applies the following steps according to the group each source belongs to.

Global Sources

These are meant to affect all screen pixels, implying that the effects' shaders must be executed on all pixels by processing a complete screen quad. Examples of these sources are big lights illuminating the entire world, such as the sun, smaller lights containing the camera, or special effects applied to the whole screen such as depth of field, fog, and so on.

Realizing that in a deferred shading system the shading cost is proportional to the number of affected pixels, global sources are one of the most fill-rate expensive. Fortunately, most of the time there are just a few of them.

These sources don't allow any specific optimization and heavily depend on how optimized the shaders being used are. In general, the individual stage performs the following steps for the global lights. For each global source:

1. Enable the appropriated shaders.
2. Render a quad covering the screen.

Local Sources

Unlike global sources, these only affect regions of the scene. Thus, any source whose influence is restricted to just a world sector falls under this category. Classic examples include small lights spread over the scene, effects such as volumetric fog, or heat haze, among others.

The benefit of local sources is that only the pixels that are inside the source bounding object have to be processed. Application of the shader only on those pixels can be implemented in several ways. The easiest is to render the source bounding object and discard back-faced polygons since bounding objects are generally convex volumes.

Even when using this approach, and almost all of the pixels are influenced by the shader, it can be slower. Applications that are vertex-transform bottlenecked and send the polygons that shape the bounding object to the graphics pipeline can result in a slower performance than only rendering all the screen pixels. This is especially true when using sphere bounding objects that are composed of a lot of polygons.

Another approach to shading only the pixels influenced by the bounding object involves rendering a full screen quad, just like for global sources, but enabling clipping and rejection features to discard as many noninfluenced pixels as possible should result in less of a performance impact.

The scissor test can be enabled using the rectangle that surrounds the screen projection of the source bounding object that was computed in the social stage. While the scissor test quickly rejects pixels at the fragment level, clipping planes can also be used to further restrain the pixels that get into the fragment stage from the transform level.

Using the scissor and clipping tests allows the system to quickly reject the pixels that are out of the projected bounding object area. The defined bidimensional region can still be very different from the real bounding object projection, and all pixels in that area are going to be shaded even if they are in front of or behind the bounding volume, as shown in Figure 2.5.4.

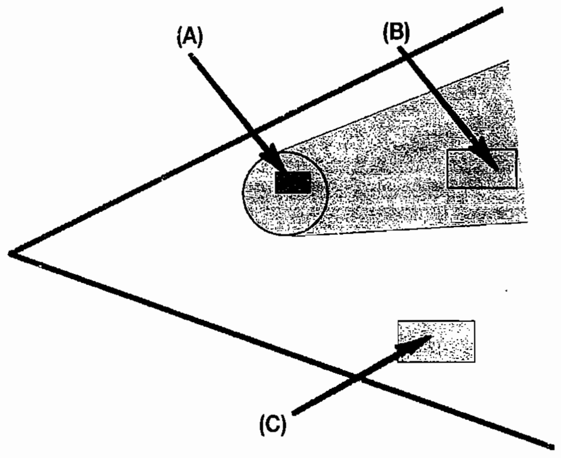

FIGURE 2.5.4 Pixels in front of and behind the bounding volume. **(A)** Inside screen projected volume. **(B)** Inside screen projected volume but not influenced. **(C)** Outside screen projected volume.

Dynamic branching can be used to discard all those pixels at the shader level. In the shaders of the lighting phase, if the pixel position is not found to be in the influence area of the bounding object, which is being passed through the shader registers, then the pixel is discarded and no lighting or effects are processed.

When dealing with spheres, the individual stage passes the sphere radius and center to the shaders, and if they find that a pixel's distance to the center is greater than the radius, the pixel is discarded. For axis-aligned bounding boxes, the process is even easier and only involves checking if the components of the position are inside the box.

The drawback is that dynamic branching is not widely supported. However, the stencil buffer can be used to emulate the dynamic branching behavior on systems that don't support it. The approach uses a cheap extra pass to create a mask on the stencil buffer, using a given condition that in this case should be if the pixel in the G-buffer is inside the influence area or not. The technique is described in [ATI05].

Using the previously described optimizations reduces the number of pixels processed and the fill rate of the deferred system. However, fill rate also relies on the processing cost of the shaded pixels.

Another nonconservative optimization that can be applied is to use a sort of level of detail to decide how many instructions each pixel uses to compute the source effect, keeping in mind that farther and smaller sources don't have to be computed with the same quality than closer sources, so it is possible to save some instructions on them.

Figure 2.5.5 shows a source rendered with three levels of detail according to how far the source is from the camera. In the first level the light is computed using the diffuse and specular components, while in the last level the specular component is not computed. However, owing to the distance at which the source is rendered, the resulting image is not distinguishable from its counterpart that renders the source with the full equation at all times.

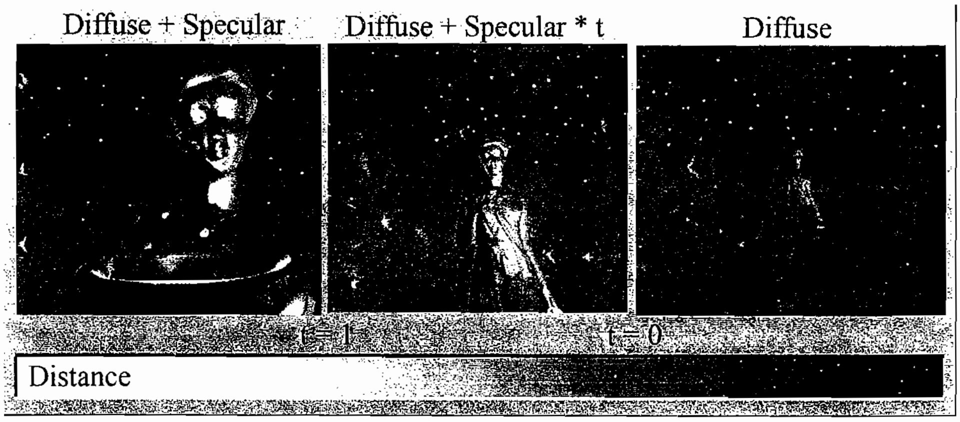

FIGURE 2.5.5 Level of detail.

When removing components from the effects equations, some artifacts and sudden popping can appear at the points where the source transitions from one level to the next. To avoid those artifacts, transition levels can be inserted that blend the previous and next level to make a smooth transition. In the presented example, the transition level multiplies the specular component by a factor that starts off being 1 to fully show the specular contribution and gradually fades to 0, fading according to the distance.

Using the presented optimizations, the individual stage of the high-level manager can be outlined as follows. For each local source:

1. Select the appropriate level of detail.
2. If dynamic branching is not supported, render a mask in the stencil buffer.
3. Enable and configure the source shaders.
4. Compute the minimum and maximum screen cord values of the projected bounding object.
5. Enable the scissor test.
6. Enable the clipping planes.
7. Render a screen quad or the bounding object.

Other Performance Optimizations

Even when the above solutions have being used to reduce the memory footprint and fill rate, they are likely to improve the general performance as well. For instance, using less memory to store the G-buffer allows a better use of the texture cache, which reduces texture transfers, thereby reducing the performance impact of packing the pixel attributes in the buffer.

This has the same effect on fill-rate reduction, which is directly bound to performance. It eliminates the fill-rate bottleneck, which is the most frequent problem in high-end graphic applications. The reduction of the fragment unit usage allows computing more complicated effects and more sources.

Together with the presented optimizations, the deferred shading system can be further optimized. As has been shown, the individual stage of the high-level manager scans all sources that need to be rendered and configures them to reduce its processing cost.

However, the sources' bounding object or a full screen quad has to be sent to the pipeline for each source. On sources that are going to be computed with the same shaders, it's possible to batch the polygons. These batched polygons can then be processed with just one draw call. They can even be collapsed into several source shapes and processed with a single shader that computes the contribution of more than one source.

When setting sources' shaders, call states can be minimized by first finding those shaders that are going to be used for each source and sorting the sources according to those shaders. Then to minimize state changes, each shader is set and all sources that use it are processed without changing the shader state.

Further nonconservative optimizations can be applied when performance is dropping. For instance, the texture that stores the pixels' colors in the G-buffer can be rendered at full size, but the other textures can be stored at lower resolutions. This way, the lighting and postprocessing phase could be computed to a render target that is half or one-quarter of the viewport dimensions and then modulated by the full-resolution color texture.

When the performance falls to unacceptable levels, lights and other effects can be computed at half or lower screen resolution while still maintaining good results if the color buffer is still rendered at full resolution. The quality loss is acceptable, given that there's a big reduction in fill rate, and therefore it is possible to add more lights and effects to modulate the scene.

While the above solutions enhance the performance of the lighting and postprocessing phase, other optimizations are applied to the geometry phase where the G-buffer is filled. For instance, when processing the scene geometry to fill the G-buffer, lights, fog, and all other unneeded effects at that stage should be disabled.

Also, it's possible to batch the scene geometry. In this instance, how the scene is batched doesn't have to match the batching used during normal rendering. This way, two different representations of the scene can be used, both presenting the same geometry but batched differently. One representation is used to render the color texture in the G-buffer. It maintains a group of polygons for each texture, and each group can be rendered with just one draw call.

The other representation is used for rendering position and normal into the G-buffer. There's no need to separate the textures and the complete geometry, so the batch can be submitted to the pipeline with just one draw call.

In practical situations the batching also has to account for occlusion and visibility, so there should be more groups to represent visibility sectors, objects, or any other visibility structure that can be used to cull out nonvisible objects and effects. However, when it's possible to keep both geometry representations in memory, the performance increases by decreasing the draw calls and other state changes, such as vertex buffer switching.

When multiple render targets are not supported, or when there are not enough to render all the textures composing the G-buffer in one pass, several passes should be done. The order in which those passes are performed can also impact performance. Before rendering the first pass, the depth buffer should be cleared, and while the rendering is performed, the buffer is filled. This first pass can suffer from overdraw, but subsequent passes use the depth values stored in the z-buffer to avoid overdraw.

The cheapest pass should be executed first to handle overdraw with the lowest penalty, and then the more intense passes should be run with the benefit of not reprocessing the same pixels. The pass that stores the normal components and the material ID should be performed before the pass that stores pixel colors from the polygon texture, vertex color, and so on.

Other possibilities are to combine forward lighting and effects with the deferred system. The per-vertex influence of some lights should be written while filling color values in the G-buffer. This technique can be used when the geometry is highly tesselated or light sources are far or don't rely on complicated lighting calculations. The difference to the per-pixel version is not very noticeable, improving the performance when the per-vertex solution can be used.

This approach can enhance performance in some situations and reduce it on others, depending on where the application bottleneck is located. The geometry properties of the surface and how many pixels compose the geometry can also influence performance, so this method should be used carefully.

Also bear in mind that most general shader optimizations can be applied to further enhance performance in a deferred shading system. Typical examples are the use of half data instead of floats when the extra precision is not needed. For materials ID and normal components, they can be stored at lower precision, and fetching the value from a lookup texture can compute the attenuation factors.

Transparency

The Alpha test is easy to integrate with a deferred shading system, but alpha blending requires several pixels to be shaded and combined. However, the G-buffer only stores information about a single pixel in each texel, so blending on a deferred shading system is not as simple as on forward rendering.

Still, some hacks can be done to allow blending on a deferred shading system. The easiest is not doing deferred rendering on polygons that need to be blended. In a first pass the application can perform the deferred path for solid polygons and then forward rendering on all the transparent polygons with alpha blending enabled.

Another approach is the rarely used technique called *screen door transparency.* It uses a stippling pattern to mask the transparent polygons so that some pixels of the background can be seen through the mask. For instance, a pattern used to represent 50% transparency will skip all the even pixels in one row and all the odd pixels in the next. When the stippling pattern is applied to a polygon, the background can be seen through the masked pixels. The holes are so small that they aren't picked up by the eye, and the eye blends the nearest pixels, giving the illusion of a transparent polygon.

Screen door transparency can be implemented directly on the deferred shading pipeline and doesn't require depth sorting. However, the screen resolution needs to be relatively high to hide the masking pattern. To make it even harder to spot the mask, the pattern can be changed and offset every frame, which also has the advantage of producing better-looking results when the transparency has a high depth complexity.

Other approaches can use depth peeling to break blended complexity into layers, which then can be rendered and blended one after the other by using a deferred shading path. This technique involves executing the complete deferred pipeline for each layer, from filling the G-buffer to source shading, which can seriously impact performance.

Anti-Aliasing

The deferred shading system can suffer from aliasing problems since the anti-aliasing pass has to be performed after the accumulation is done in the lighting and postprocessing phase.

Even when the graphics hardware can natively support multisampling in the G-buffer, lighting and other effects can create artifacts near the polygon edges. This happens because there's no multisampling support on the shading phase and because the work is performed in image space. When dealing with floating-point textures composing the G-buffer, the graphics card may not even support multisampling when writing to the G-buffer.

Several solutions can reduce the aliasing problems on a deferred shading system. The easiest approach is to use over-sampling. That technique is implemented by performing the entire rendering at higher screen resolutions and then blurring down to the desired output resolution. For instance, the rendering can be performed at 1600 × 1200 and then in a final pass scaled down to 800 × 600 using a bilinear filter.

Over-sampling has the advantage of noticeably reducing the aliasing artifacts in the scene; however, it requires more memory to store the G-buffer. This heavily increases the application fill rate, becoming prohibitively expensive in most applications.

Another solution for implementing anti-aliasing is to use an edge-smoothing filter [Fabio05]. This filter is implemented as another source effect, which is introduced as the last source in the postprocessing phase of the deferred pipeline. The filter performs two steps.

First an edge-detection scan is applied to the screen. To perform the edge detection, the filter uses the discontinuities in the positions and normal stored in the G-buffer. The results can be stored in the stencil buffer as a mask for the next step.

Later the screen is blurred using only the pixels that are edges. This is accomplished by performing a blurring only on the pixels that are masked in the stencil buffer. A by-product of the blurring is that the color will bleed into the edges of the polygons. This can result in the background bleeding into the character, producing undesired results. To eliminate the color bleeding, the blurring process should be refined. A kernel is applied to the edge pixels, but instead of blurring all the pixels that lie in the kernel, only the closest to the camera are combined.

This way the closest polygons are smoothed and no color bleeding will happen when the polygons are overlapped. ATI [ATIGDC] has a more in-depth description of the color bleeding reduction algorithm. It is oriented toward depth of field, but the process is the same.

This technique allows the reduction of the aliasing artifacts without increasing the memory footprint required to store the G-buffer. Even when the fill rate is increased, the use of the stencil buffer to quickly discard the nonedge pixels on the second step helps reduce the fill-rate impact of the blurring. This technique can be disabled if there is a severe performance drop and dynamically enabled again when the performance gets back to a tolerable level.

Another way to simulate anti-aliasing is to perform a separate pass when storing the color in the G-buffer. In this pass we don't render the colors to a floating-point texture, but to the classical RGBA8 frame-buffer with anti-aliasing enabled.

This way, only the colors receive anti-aliasing; lights and effects don't. When lighting is not anti-aliased, the results can be acceptable, but when colors are not anti-aliased, the results are not good.

It's also possible to combine the color anti-aliasing with the edge detection filter, blurring, and color bleeding reduction for the lighting and effects, which improves the quality of the resulting anti-aliasing image.

Conclusions

Deferred shading is a nice solution to deal with multiple lights influencing a scene. It keeps everything simple and separated and allows handling next-generation scenes with a high number of polygons, complex materials, and lots of special effects and lights.

However, as has been shown, deferred shading has some huge disadvantages such as memory use, high fill-rate, and the inability to handle transparency nicely, and most implementations suffer from aliasing problems.

This article has presented several techniques to reduce each of these drawbacks. Proper planning in the low-level pipeline, with respect to the shader implementation, and use of G-buffer capabilities help improve performance. The high-level mangers in the application layer and general optimizations increase the performance and scalability of the system. Using all these techniques makes it possible to overcome or manage most of the major drawbacks.

Most of the presented solutions can be controlled in real-time, which allows for adjusting the image quality according to the average frame rate. This helps maintain a constant frame rate, which in the end plays a major role in the user immersion experience.

When combining these techniques and the potential of a deferred shading system to handle next-generation scenes simply and fast, it makes a very attractive solution for highly detailed graphics on current and next-generation systems.

References

[ATI05] ATI, "Dynamic Branching Using Stencil Test." ATI Software Developer's Kit, June 2005.

[ATIGDC] ATI, "Advanced Depth of Field." Available online at *www.ati.com/developer/gdc/Scheuermann_DepthOfField.pdf.*

[Buttza05] Butterfield, Ryan. "Packing a Float into a Texture." Available online at *http://www.gamedev.net/community/forums/topic.asp?topic_id=322318&whichpage=1&#2078865.*

[Calver03] Calver, Dean. "Photo-realistic Deferred Lighting." Available online at *http://www.beyond3d.com/articles/deflight/*, July 31, 2003.

[Delphi3D] "Deferred Shading." available online at *http://www.delphi3d.net/articles/viewarticle.php?article=deferred.htm*, October 24, 2002.

[Fabio05] Policarpo, Fabio and Francisco Fonseca. "Deferred Shading Tutorial." Available online at *http://fabio.policarpo.nom.br/docs/Deferred_Shading_Tutorial_SBGAMES2005.pdf*.

[Geldreich04] Geldreich, Rich. "Gladiator Deferred Shading Demo." Available online at *http://www.gdconf.com/conference/archives/2004/geldreich_rich.ppt*, March 16, 2004.

[NVIDIA04] Hargreaves, Shawn and Mark Harris. "Deferred Shading." Available online at *http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf*, March 26, 2004.

[Pritchard04] Pritchard, Matt. "Deferred Lighting and Shading." Available online at *http://www.gdconf.com/conference/archives/2004/pritchard_matt.ppt*, March 7, 2004.

[Puig05] Puig, Frank. "Deferred Shading Demo." Available online at *http://fpuig.cjb.net*, February 15, 2005.

[Puig06] Puig, Frank. "Fast Per-Pixel Lighting with Many Lights." *Game Programming Gems 6*, edited by Michael Dickheiser. Charles River Media, 2006.