Deferred Rendering

Jonathan Thaler*
TU Wien

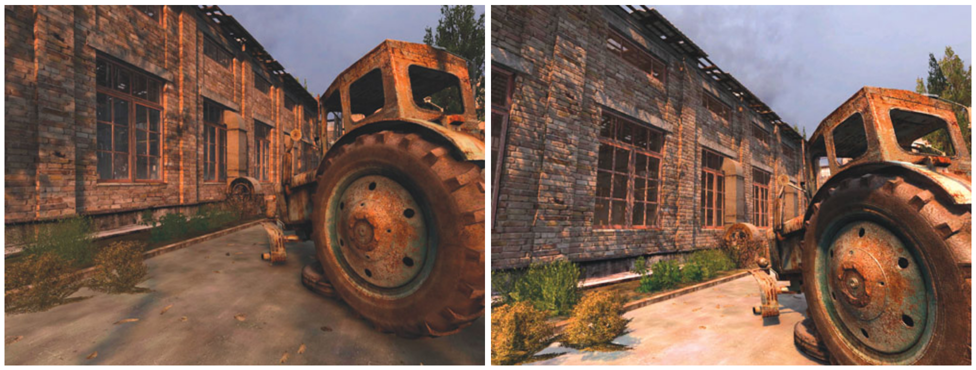


Figure 1: Forward rendering (left) compared to deferred rendering (right). Image courtesy of GSC Game World.

Abstract

In this paper an overview of the deferred rendering technique and its applications is discussed. First, a short review of forward rendering and its drawbacks is given. Then the paper dives into a technical overview of the deferred rendering technique and its implementation together with the pros and cons involved. A short review of its applications in different areas like games and scientific visualization follows. Furthermore an overview of different effects and techniques easy to achieve with deferred rendering as opposed to forward rendering is presented. Finally a discussion on parallel techniques for deferred rendering and an outlook on extensions and alternatives is presented.

Keywords: deferred shading, deferred rendering, rendering, realtime graphics

1 Introduction

Until some years ago forward rendering has been the state-of-the-art approach in a realtime rendering engine. Some scene description is traversed to generate a render list of potentially visible objects which are then rendererd with their shaders, effects and lighting. In the rendering process the geometry is sent through the vertex shader and the outcoming pixels are then shaded during one or more passes in a fragment shader. The most expensive part of shading is naturally the process of lighting which can be implemented in two ways in forward rendering.

- Single pass. For each object all affecting lights are searched and then all lighting and material for an object are rendered in a single shader.
- Multi pass: For each light iterate over each object and add the contribution of the objects lighting to the framebuffer.

*e-mail: e0225907@stud4.tuwien.ac.at

The following drawbacks of forward rendering can easily be identified

1. Complexity of rendering the lighting is $O(Lights * Objects)$.
2. Because each material needs a specific shader for each light-type (point, directional, spotlight,...) the numbers of shaders to be maintained rises very fast (combinatorical explosion). Shaders require sometimes multiple passes to calculate lighting so passes increases further.
3. When using the single pass approach all shadow maps (one for every light) needs to be hold in memory which can lead to a bottleneck very quickly.
4. State changes increase darmatically with lighting. The current limit is about 250 changes per frame on current hardware.
5. Reducing the overdraw - already shaded objects get covered by others - to zero is virtually impossible so forward rendering inherently wastes shading.

With the advent of Shader Model 3.0 together with more powerful hardware like the GeForce 6800 around 2004 it became possible to address these drawbacks of forward rendering by utilising deferred rendering. The technique is not a new one and dates back in 1988 where it has been proposed by [Deering et al. 88] although they didn't use the term deferred rendering.

The next section gives an overview of the technical details involved with deferred rendering and also discusses the drawbacks and performance issues of it. A more in-depth tutorial on implementing deferred rendering can be found in [Policarpo and Fonseca 2005] and [Hargreaves and Harris 2004]. [Shishkovtsov 2005] and [Koonce 2008] show a more high level approach to it and discuss advanced issues.

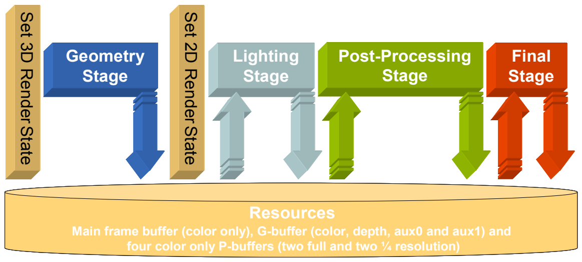

Set 3D Render State
Geometry Stage
Set 2D Render State
Lighting Stage
Post-Processing Stage
Final Stage
Resources
Main frame buffer (color only), G-buffer (color, depth, aux0 and aux1) and four color only P-buffers (two full and two ¼ resolution)

Figure 2: Architecture of deferred rendering. Image courtesy of [Policarpo and Fonseca 2005].

2 Implementation

Deferred rendering differs fundamently in the way that it decouples the processing of the geometry from the process of shading it. With this approach the drawbacks of forward rendering can be circumvented but different problems and pitfalls rise. Figure 2 shows an overview of the Architecture of the deferred rendering technique. The geometry of each object is rendered in a first pass which occurs in the geometry stage into the so called Geometry-Buffer (G-Buffer) without applying shading. In the light stage the shading is then applied globally to this backbuffer where all attributes needed for lighting are stored and the result is written into the post processing buffer. In the post-processing stage additional shaders can be applied to the post processing buffer to create effects explained in section 4. The final stage simply displays the result of the post processing buffer onto the screen.

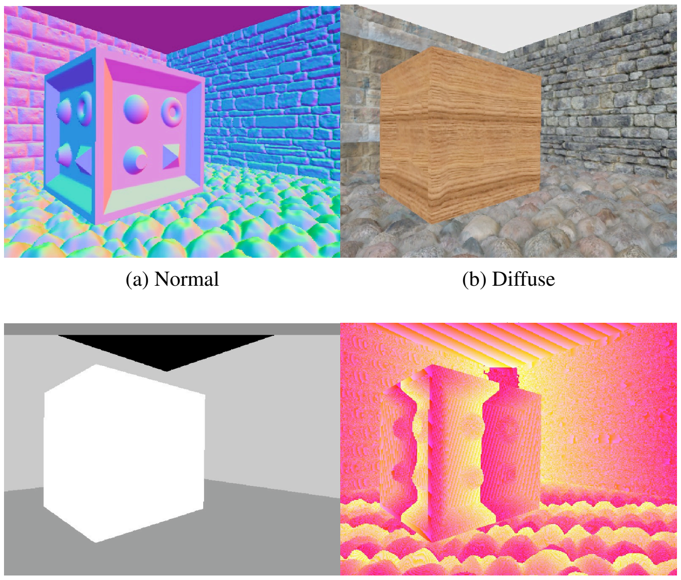
2.1 Geometry Stage

In the geometry stage the target is to render attributes of the geometry and its material needed in the lighting pass into the G-Buffer. The specific attributes differ from application to application but at least these four attributes are common to all deferred rendering pipelines.

- Normals
- Diffuse color
- Specular color and shininess
- Depth

Figure 3 shows a visualization of the content of the G-Buffer split up into each attribute.

Obviously it is not possible to store all attributes in one single buffer due to the memory needed for each attribute and limited texture formats. Therefore the geometry pass is implemented utilizing Multiple Render Targets (MRT) and vertex shaders. MRT allows the simultaneous rendering into up to four back-buffers which is exactly what deferred rendering needs. A severe restriction is that MRT must have all the same bit-depth so using 32 bit depth this leads to an example-layout for the four attributes shown in Figure 4.
Several tricks exist to reduce the amount of channels needed e.g., for normals and depths by storing only the x and y component and reconstructing the z-component by some calculations.



(a) Normal (b) Diffuse

(c) Specular (d) Depth encoded as color. Jumps in color comes from encoding.

Figure 3: Visualization of the four targets of the G-Buffer

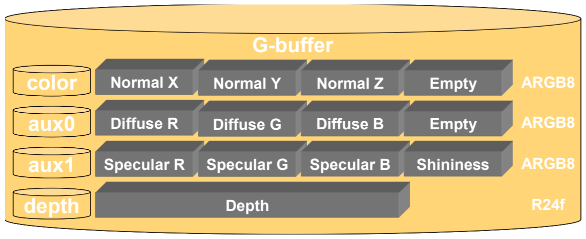


Figure 4: Geometry Buffer. Image courtesy of [Policarpo and Fonseca 2005].

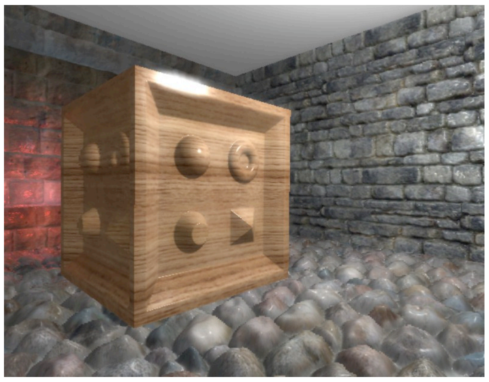


Figure 5: Final Image

2.2 Lighting Stage

As a result of the geometry pass one gets access to four textures in which the above mentioned attributes are stored. The lighting pass uses these textures to calculate the lighting. This is done by iterating over all lights which affect the geometry and additively apply the lighting equations based upon the attributes. The light volumes are transformed into screen space so their projection covers the pixels to shade. The following volumes are used.

- Pointlight: sphere volume
- Spot light: cone volume
- Directional light: full screen quad due to no falloff and the direction.

An issue is that for a sophisticated material-model different calculations are needed based upon the type of the model. This can be either implemented in one huge shader or be split up into separate shaders with an additional attribute is used to indicate the type of material. Of course in this stage there are both vertex and pixel shaders active but the pixel shader produces a different output than in a forward renderer.

2.3 Post Processing Stage

This stage takes the output of the lighting stage and the additional information from the G-Buffer to implement effects like Motion Blur, Depth Of Field and Bloom as a post processing effect. See section Effects for a further discussion of these topics in the context of deferred rendering.

2.4 Final Stage

In the final stage the result of the post processing stage is brought onto the screen by texturing a quad polygon which has the size of the screen-resolution with the output of the Post Processing Stage.

Figure 5 shows the final result of applying the lighting and post-processing pass on the above four attributes.

2.5 Drawbacks

Despite its convincing advantages over forward rendering, deferred rendering has two major drawbacks. Those can be solved by using either clever workarounds or hacks and tricks which fall back to forward rendering as shown below.

- Multisample Antialiasing MSAA and Coverage Sampling Antialiasing CSAA. Deferred rendering is inherently incompatible with MSAA and CSAA because MRTs don't support it. [Policarpo and Fonseca 2005], [Shishkovtsov 2005] and [Koonce 2008] all apply an edge-detection filter during post-processing and blur the edges to solve this issue.

- Transparency is not possible to directly render within a deferred renderer because the first hit with a surface is recorded and no multi-layer frame-buffer as proposed by [Bavoil et al. 2007] exist yet in hardware. [Shishkovtsov 2005] and [Koonce 2008] both propose forward rendering as a fallback solution for rendering alpha blended geometry after the lighting pass. Despite the drawbacks of transparent rendering, with the depth values at hand gathered through the geometry pass, interesting effects like a more sophisticated water and refraction effect can be implemented as shown in section 4.1.

2.6 Performance

Although deferred rendering has many performance benefits opposed to forward rendering there are some points which need careful consideration and experimentation otherwise a deferred renderer could very easily suffer from bad performance.

- **G-Buffer layout and format**. If using fat formats e.g. 64-bit depth this can mean that reading a texel can take twice as much time. Another point to consider when designing the G-Buffer layout is how much performance one wants to spend on unpacking e.g. when storing only the x- and y-component of the normal. [Shishkovtsov 2005] gives an overview of different G-Buffer layouts, [Valient 2007] discusses the highly packed G-Buffer layout used in the game Killzone 2.

- **Bound lights**. A naive approach of deferred rendering would apply the lighting calculations to all pixels on the screen but this is only necessary for global directional lights. For most of the local lights their influence on the pixels can be bound and so the number of pixels to be processed by the shader. Different techniques like early z-rejection, stencil masking and dynamic branching exist for this approach. [Policarpo and Fonseca 2005] propose to utilize the scissors test to bound the influence of each light. The bounding rectangle of a light is calculated in screen space on the CPU and then passed to OpenGL for the scissor test. [Koonce 2008], [Valient 2007] and [Hargreaves and Harris 2004] propose light-volumes by using stencil mask to mark pixels which are really affected by the light.

- **Multiple Materials**. In deferred rendering one shader needs to cover the lighting of multiple materials. One solution as implemented by [Shishkovtsov 2005] is to store the material-id in the G-Buffer and do dynamic branching in the shader. Another approach is to render the light for each material it affects but this would waste most of the deferred renderings benefits. [Hutchinson et al. 2010] propose a technique to split the screen into tiles of different classifications like sky, MSAA edge, sun-facing and soft-shadow. They generate then a shader id out of the classification which allows a more specific rendering.

2.7 Benefits

The use of deferred rendering turns out to excel in these areas in which forward rendering has difficulties.

- Decoupling geometry from shading and thus reaching a complexity for shading of $O(Objects + Lights)$.
- Much lower shader complexity and no combinatorial explosion.
- Much less wasted shading than in forward rendering.
- A kind of global information in the form of the G-Buffer which can be utilized in huge variations as shown in the sections Effects and Advanced Techniques.

3 Applications

3.1 Games

The most prominent application for deferred rendering are games. In the last years since the upcoming of powerful enough hardware deferred rendering found itself being implemented in a number of AAA Titles like the S.T.A.L.K.E.R. series, Tabula Rasa, Killzone 2 and Starcraft 2. [Filion and McNaughton 2008] and [Valient 2007] give an in depth discussion of their implementation of deferred rendering.

The question is why not all modern games use deferred rendering as their primary rendering technique and the answer is simply that not all type of games and scenarios are equally well suited for it. According to [Hargreaves and Harris 2004] deferred rendering is very well suited for complex scenes with many local lights which ideally don't overlap. In this case the lights can be bound very well to a small amount of pixels they influence in the G-Buffer and thus reducing shader complexity to $O(R)$ where R is the screen resolution. Many directional lights are the worst scenario for deferred rendering because they require to run their shaders over the complete G-Buffer and not only a small subset of pixels. There shading complexity raises to $O(R * Lights)$ where R is the screen resolution. Applied to games this would imply nighttime scenes with many dynamic lights as ideal for deferred rendering and outdoor scenes with one ore more large directional light source like the sun as the worst scenario. An exceptional scenario when to use deferred rendering is when there is no tight control over the environment, its effects and thus the count of local lights.

As an example Starcraft 2 as a realtime strategy game is ideal for deferred rendering because a huge amount of units can combat each other and could cast a vast number of dynamic lights. The game develops some episodes of it's story line in in-game graphic cutscenes which use highly complex effects and techniques (see section 4) so deferred rendering was a good choice because these scenes benefit a lot from it.

Another example is Tabula Rasa which is a MMORPG and thus inherently has no direct control over the environment. A potentially unlimited amount of players can logon and combat each other or the environment and thus increase the complexity for rendering. It has a day-night cycle which in night creates an ideal scenario for deferred rendering.

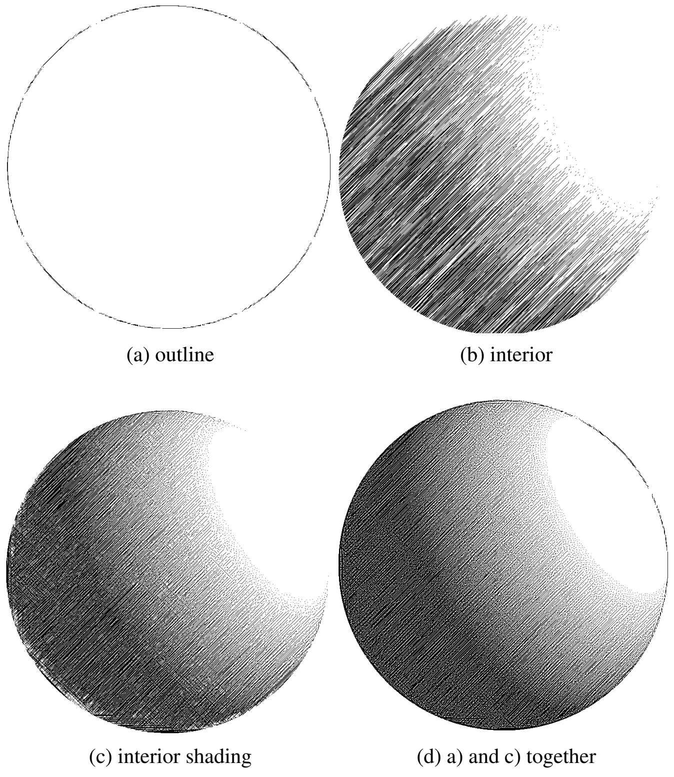


Figure 6: Different styles of 2+D NPR technique

3.2 Scientific Visualization

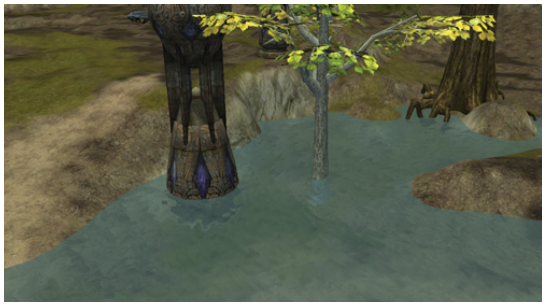
Deferred rendering can also be applied to volume rendering as proposed by [Treavett and Chen 2000]. In their paper they show a pen-and-ink style rendering technique of volumes to do non photorealistic rendering (NPR) to achieve volume illustration effects. To overcome the difficulties of just having access to very local data around a voxel during rendering they suggested a two-phase approach. In the first phase all 3D information in object space is gathered and in the second phase the NPR effects can be applied globally in image space. Although they call it 2+D NPR technique because of the access to a subset of spatial information they never mention the term deferred rendering but it is exactly the same procedure like deferred rendering. See Figure 6 for some results rendered with this technique.

4 Effects and Advanced Techniques

With the use of global depth and other pixel informations, global effects become possible. In this chapter some of the most common effects to deferred rendering are presented and some details discussed.

4.1 Water and Refraction

Direct water and refraction rendering is not possible with deferred rendering due to its incompatibility with alpha blended geometry. [Koonce 2008] brings in a forward renderer in combination with


(a) Forward rendering

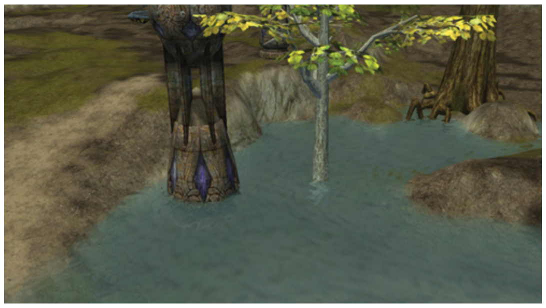

(b) Deferred rendering

Figure 7: Water with refraction rendering compared. Image courtesy of Destination Games.

deferred rendering to do water rendering in Tabula Rasa. See Figure 7 for a comparison of the water renderer in Tabula Tasa with only forward rendering (a) and with deferred rendering (b). One can see that the color of the water smoothly darkens in the deferred rendering version whereas in the forward rendering sharp edges at the shore of the water show up. This effect was implemented by accessing the G-buffers depth values in the water shader of the forward renderer. An additional artist-crafted 1D-texture was brought in to give artists the control over the gradient of the water color.

4.2 Depth of Field

Depth of field is the effect where objects appear in focus and objects nearer or farther away from the viewer appear out of focus which results in the blurring of those objects. This effect is quite natural in photography and film making and is used as a tool for directing the attention of the viewer.

Depth of field is caused by light reflected by an object falling through a lens and being not projected directly on the plane in focus. The bigger a lens is the bigger gets the circle of confusion and vice versa. See figure 8 for an illustration of the circle of confusion. The object lies not on the plane in focus and is therefore projected behind the image plane which causes the blurriness. The same is true for objects with a higher distance to the lens than the plane in focus. Their rays will meet in front of the image plane and thus resulting again a blurred object.

Because realtime-rendering does not simulate the light distribution by shooting rays through a lens like a sophisticated ray-tracer [Pharr and Humphreys 2004] but by performing scan-line rendering and thus behaving like a pin hole camera this effect is not naturally built in with OpenGL or DirectX and needs to be simulated with other techniques.

[Demers 2004] cites five different methods of generating the depth of field effect.

1. Distributing traced rays. Tracing rays in realtime-rendering despite the ever increasing performance of hardware is still not acceptable at interactive frame rates but would lead to the correct solution.

2. Rendering from multiple cameras - accumulation buffer technique. In this technique the scene is renderer multiple times into an additional back buffer - the accumulation buffer - each time the camera a little bit altered - and the resulting pixels are then blurred. Due to the need for very much passes for acceptable quality (around 50) this technique is not amenable to realtime-rendering.

3. Rendering multiple layers. In this technique the scene is divided into multiple layers and each layer is rendered blurred based on the distance to the image plane.

4. Forward-mapped z-buffer technique. This technique scatters color from a pixel to its neighbors .

5. Reverse-mapped z-buffer technique. First multiple mip-map levels of the resulting image are calculated and then based upon the circle of confusion of each texel the according mip-map is chosen. Because the smaller mip-maps are scaled to the original solution this results the blurring.

The last two techniques introduce the basic concepts of rendering depth of field in a deferred renderer. They utilize the z-value which are already at hand in deferred rendering through the G-Buffer and they either gather or scatter color from or to neighboring pixels. All practical real-time depth of field implementations in deferred rendering build upon these three concepts.

A plain forward approach would be to calculate each pixels circle of confusion and use this factor to scale a blur kernel (e.g. a poisson disc) which operates over the whole image in the post-processing stage. Unfortunately this just works for objects beyond the focal plane, doesn't scale well to low-end hardware and causes artifacts like pixel bleeding as shown in Figure 9.

A highly specialized implementation of the depth of field was developed for the game Starcraft 2 and has been discussed by [Filion and McNaughton 2008]. Instead of applying a blur filter three images of different levels of blur are generated and interpolated to achieve a gradual blur. The circle of confusion is calculated and stored for every pixel in a separate texture. Then blurred circle of confusion and blurred depth maps are calculated. With this information at hand depth ordering is used to do correct pixel bleeding-free depth of field. The depth ordering works like this: Compare the average depth of the filtered area to the current processing pixels depth. If the depth of the current pixel is higher then it is behind its neighbours and thus the circle of confusion value of blurred circle of confusion map is used. If the current pixel is nearer then it is in front of its neighbors and in this case the circle of confusion is computed and used just for this pixel and nothing from the circle of confusion map.

Another implementation of depth of field for the Game Call of Duty 4: Modern Warfare was shown by [Hammon 2008]. Although they don't utilize a deferred renderer they generate a depth map - as present in deferred rendering - during rendering and apply depth of field as post process by calculating different circle of confusion maps and interpolating between them.

See figure 10 for a difficult depth of field scene in the game Starcraft 2. In this scene a sharp character in the plane of focus appears in front of the out of focus and thus blurred background. Another

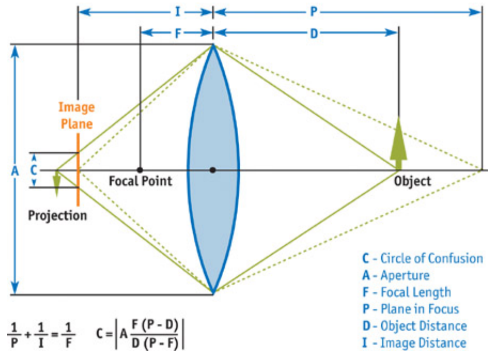


Figure 8: Circle of confusion

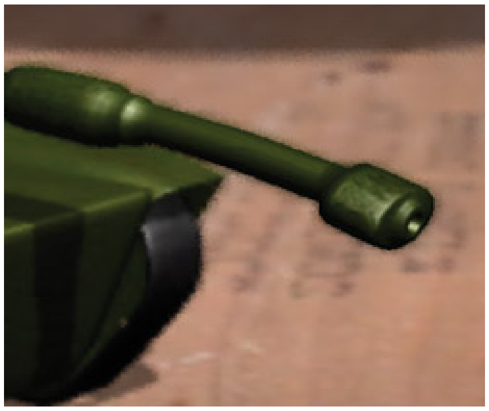


Figure 9: Pixel bleeding

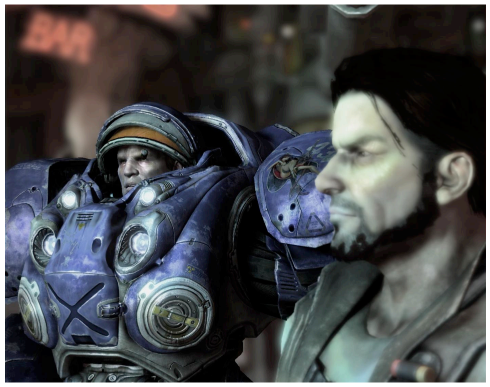


Figure 10: Difficult DOF scene. Image courtesy of Blizzard Entertainment.

character appears in front of the sharp one out of focus and is therefore blurred too.

4.3 Global Methods

With the information of depth and normal for each pixel a number of global effects become possible. First a short overview of how depth and normals can be used for global calculations and then three global methods are introduced and their issues are discussed.

The approach to do global effects is to sample neighbors nearer or farther away with an arbitrary number of samples. This can be accomplished by ”shooting” rays as shown in figure 11. Of course we are not in real 3D but in image-space when sampling the G-Buffer and rays must be projected back into 2D to access the neighbors on the G-Buffer. In a real global illumination ray-tracer as presented by [Pharr and Humphreys 2004] BRDFs are used to calculate the distribution of the rays over the hemisphere of a point but this is not feasible for real-time rendering. Instead the assumption is made that the surface is perfect diffuse and all rays are equally distributed over the hemisphere. So this implies that global effects won’t work in this way for other surfaces then lambertian like perfect specular material.

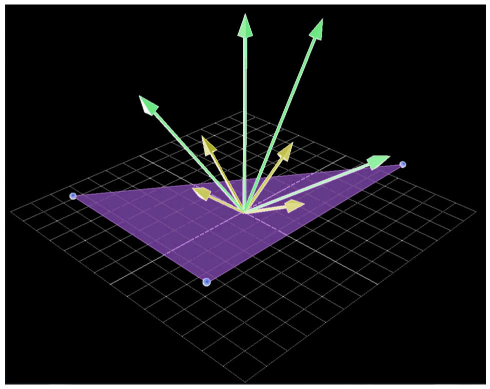
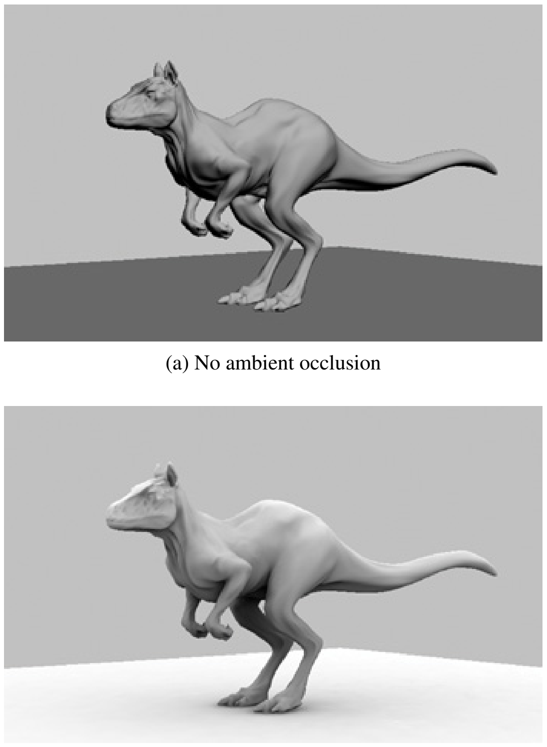


Figure 11: Shooting rays for neighbor sampling. Image courtesy of Blizzard Entertainment.

4.3.1 Ambient Occlusion

Ambient occlusion is an effect very easy to understand and although it makes up very much of a correct visual impression of a 3D scene



(a) No ambient occlusion

(b) With ambient occlusion

Figure 12: Comparison of rendering with and without ambient occlusion. Image courtesy of [Pharr and Green].

it still has been very difficult until today to integrate it into real-time graphics. Ambient occlusion is simply the result of geometry occluding each other in an indirect way and thus blocking incoming indirect light. See figure 12 for a comparison of ambient occlusion and standard flat-rendering. Notice the complete different shading of the kangoroo especially below it and around its stomach. In this picture there is a very subtle shadow casted by the kangoroo on the floor which is also caused by ambient occlusion. [Pharr and Green 2004], [Bunnell 2005] and [Jared and Jia 2008] give an overview of different implementations of ambient occlusion.

For deferred rendering a so-called screen space ambient occlusion (SSAO) technique has been invented. See figure 14 for an example of it. Notice how occluded areas darken at any distance. The idea is to sample the depth of the neighboring pixels in screen space. The depths of the pixel from where samples are cast and the depths of the samples are compared. If the samples depth is lower there is a surface near to the origin pixel and ambient occlusion may be occur. Whether ambient occlusion really occurs or not depends on the occlusion function which must be carefully designed. See figure 13 which shows an example of an occlusion function. Notice the small epsilon which is used to handle special cases. Some care must also be taken to prevent self-occlusion which simply means rays which point below the surface are e.g. flipped or ignored. SSAO is normally not able to render the subtle shadow as shown in figure 12b because this would take too much samples at a too far distance of neighbors and is more a global illumination effect than ambient occlusion. In fact in real global illumination there is no distinguishing between global illumination and ambient occlusion because a real global illumination algorithm brings ambient occlusion with it for free.

[Filion and McNaughton 2008] discusses the implementation of

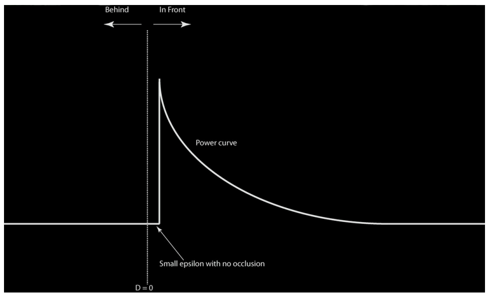


Figure 13: An occlusion function. Image courtesy of Blizzard Entertainment.

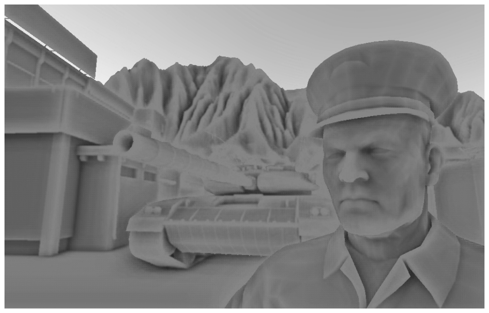


Figure 14: Screen Space Ambient Occlusion. Image courtesy of Crytek.

ambient occlusion in Starcraft 2. They use a random 3D vectors texture which allowed to create the random sampling pattern for each pixel. Of course randomized sampling is not cache friendly and the wider the area to sample the worse it gets. As another performance issue textures have been reported to be a bottleneck. To increase performance a lower resolution depth-map with quarter size was used.

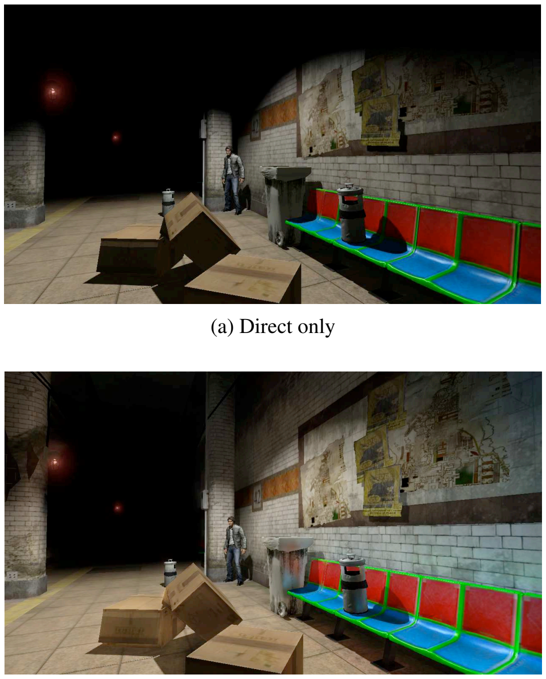
[Mittring 2007] give a short introduction of how SSAO was implemented in CryEngine 2 used in Crysis and their approach was basically the same as the one in Starcraft 2.

4.3.2 Global Illumination

[Filion and McNaughton 2008] report that they implemented global illumination the same way as their ambient occlusion technique just by sampling a wider area.

[Soler et al. 2010] calculate global illumination in deferred rendering by first mip-mapping the G-Buffer and then calculating the indirect lighting at each level. This calculation is based upon Monte-Carlo sampling to get uniform screen-space samples. The result is the combination of all contribution from all mip-map levels. See figure 15 for a comparison of direct only lighting and direct with one indirect bounce lighting. Their approach is real-time compatible as the picture from figure 15b took 10 ms to render.

[Shishkovtsov 2005] reported that during the development of S.T.A.L.K.E.R. experiments of extending the deferred renderer to support global illumination of diffuse surfaces have been made. This idea was then of course dropped due to insufficient performance although it ran about 10 fps with 500 indirect lights.



(a) Direct only

(b) Direct and indirect

Figure 15: Comparison of rendering with and without indirect bounce. Image courtesy of Eden Games.

4.3.3 Subsurface scattering

This has been a very difficult technique to implement in real-time so far because of the lack of global information and its inherent very costly process. Even global illumination ray-tracers use simplified models like the diffusion-model to do subsurface scattering. Although there exist different approaches for real-time rendering as presented by [Hao and Varshney 2004], [Hao et al. 2003], [Carr et al. 2003] and [Green 2004] with the approach of SSAO at hand these methods could be adapted to implement subsurface scattering in deferred rendering.

4.4 Other Techniques

Beside the above mentioned techniques there exists a vast number of other techniques which become more suitable with the use of deferred rendering. Here some other techniques worth mentioning which can be improved using a deferred renderer are presented.

- **Motion Blur**. This technique is easy to implement using deferred rendering. The motion vectors of each object of the scene are rendered into additional channels in the G-Buffer. In the lighting stage then sampling along direction of the vectors is performed to achieve the directional motion blur. See [Rosado 2008] and [Ritchie et al. 2010] for more details.

- **Shadowing**. Deferred rendering works both for shadow maps and stencil shadows although the later is hardly used anymore due to its hard edges and high fill rate consumption. The implementation for soft shadow maps can be increased in quality with the use of additional data at hand in deferred rendering. See [MohammadBagher et al. 2010] for more details on enhancing the quality of Percentage Close Shadow Maps with deferred rendering.

4.5 Parallel Rendering

Todays games spend about 25-40% of their rendering time in the graphics driver and GPU. During this time if not special care has been taken the CPU stalls idle and just waits for the return of the call. If the game-engine is single-threaded this would implicitly delay the running of other subsystems like physics or AI. If it is possible to keep the CPU doing useful work during these calls rather than just waiting for their return a huge speedup could be gained.

It is important to consider that rendering is at its lowest level - the GPU - an inherent single threaded task because of the GPU can just process one issued command after another. But rendering also involves a large amount of CPU cost for issuing commands and setting up their data so this is the place where gain can be gained. Because deferred rendering is split into at least two passes parallelism can be exploited to reduce CPU stalling as much as possible.

One way to approach a parallel rendering is to separate independent engine-subsystems into different threads. In this case a stalling in one subsystem doesn't have an influence on all the others. Unfortunately in game-engines there is hardly a really independent subsystem because everything relies on each other. The physics produce in conjunction with the animation-system the new orientation matrices which gets passed to the renderer somehow and the sound-system must also be tightly synced with those subsystems to prevent out-of-sync effects. As one can see this approach would need at some point one or more sync-points where the subsystems can exchange data. If this locking is not very clever this would bring the whole system to a grind. See [Tulip et al. 2006] for a more in-depth approach on the design of a multithreaded game-engine.
Another approach divides the work that has to be done into fine-grained batches which get distributed over the threads but for this the algorithms used must support parallelism. A perfect example for such an algorithm are occlusion queries used in conjunction with deferred rendering. Occlusion queries allow to start a query, issue the rendering of some primitives - normally low-poly models without any shading, stop the query and then wait for the result from the GPU. The primitives aren't really rendered but only their z-value is produced and compared to the z-values currently set in the frame-buffer. The amount of pixels which would have been potentially drawed are returned as a result as soon as the query has finished.
Because of the inherent latency and the built-in mechanism to check whether the result is already present or not occlusion queries are useful in parallel rendering. A very sophisticated algorithm that builds upon occlusion queries to do dynamic occlusion culling is presented in [Mattausch et al. 2008]. A key point in this algorithm is that within queries many geometric objects are rendered in a batch which could take some time for the result to be available. As an example for deferred rendering during the G-Buffer rendering we could theoretically also render the depth to the framebuffer to have a z-value. Then during the lighting and final stages we then could perform occlusion culling as described in [Mattausch et al. 2008] and so split the rendering in smaller tasks.

[Lorenzon and Clua 2009] discuss a straight-forward approach for deferred rendering utilizing DirectX 11. In this version of Direct3D command buffers have been introduced which allow the issuing and cleaning up of rendering commands from different threads. In this paper these command buffers are used to split deferred rendering into three parallel steps: the G-Buffer creation, lighting with non-shadow casting lights and lighting with shadow casting lights. They reported an increase in performance just at a high amount of objects which is because higher object count put more stress on the CPU to issue the commands and run through the driver. Although it is not a very sophisticated approach it gives an interesting direction to start with when researching parallel rendering for deferred shading.

A complete different approach is possible on the Playstation 3 as shown by [Heirich and Bavoil 2005]. The Playstation 3 has a very powerful hardware architecture - the Cell Broadband Engine - at its heart. It consists of 1 PowerPC Processing Element PPE and 8 Synergistic Processing Elements SPEs. Because the PPE contains 2 cores this makes a total of 9 threads of execution. The Playstation 3 draws a big amount of its power from the unified memory architecture which allows the SPEs to access memory from GPU. Although programming is tricky and introduces synchronizing primitives specific for this platform with careful design the Playstation 3 can deliver about the same throughput as todays modern hardware. In the paper by [Heirich and Bavoil 2005] 5 SPEs do the lighting calculations on the G-Buffer data which comes from the GPU. The challenge in programming the SPEs is that they must receive data in a stream of blocks. This was accomplished by sending the data of each scanline to another SPE which then run the pixel-shader on it and delivered the result to the GPU memory.

[Valient 2007] report to use the SPEs of the Playstation 3 a lot during rendering in Killzone 2 so their approach can be classified as a parallel deferred renderer too. They generate display lists, do scene graph traversal/visiblity culling, skinning, particles, ... on the them. An interesting feature is that the engine is data driven and no direct draw-calls on objects occur. The objects themself store decision-trees which contain the parts to draw together with their shaders and other stuff. The SPEs then traverse the scenegraph to find the according objects and process their decision-tree to find parts to draw.

5 Extensions and improvements

Some work has been done on the field of extending deferred rendering to overcome the limits of it not being able to render transparent objects and being incompatible with anti aliasing.

[Engel 2008] proposed the use of a light pre-pass together with an additional light-buffer. In the first pass a reduced G-Buffer with only depth and normal values is rendered. Then in the next pass light properties are rendered into the light-buffer using the reduced G-Buffer as a screen space operation. In the last pass the main geometry is rendered with full texturing etc. and using the light-buffer to calculate lighting. This approach supports full MSAA/CSAA because of rendering the geometry in a way forward rendering does.

Inferred Lighting by [Kircher and Lawrance 2009] builds directly on deferred rendering but extends it in several ways to achieve lighting of alpha blended geometry and MSAA. It introduces an additional lighting-pass in which the lighting is calculated with the inputs from the G-Buffer and the result is written to a separate L-Buffer. Both the G-Buffer and L-Buffer have a lower resolution in inferred lighting than the frame buffer. Additionally discontinuity sensitive lter DSF data is calculated and stored in the geometry stage. This data is used in the material stage to up-sample the L- and G-Buffer to calculate an unaliased result. Transparent geometry is encoded in a stipple pattern which is recognized in the material-stage and rendered correctly with the use of the DSF data.

6 Conclusion

Deferred rendering is a powerful technique to render difficult lighting situations where normal forward rendering would break. The main drawbacks of not being able to handle transparent geometry and MSAA can be overcome with straight forward tricks but extensions and newer exploits exist which solve these issues.
Most implementations are found in AAA Games like S.T.A.L.K.E.R., Starcraft 2 and Killzone 2 but it has also found its applications in non-photo-realistic volume rendering.
With the G-Buffer at hand a series of image-space effects and techniques like depth of field, ambient occlusion, motion blur and to some extend global methods like ambient occlusion, global illumination and subsurface scattering can be achieved. It is also a good candidate for parallel rendering due to its inherent multi pass technique.

References

BAVOIL, L., CALLAHAN, S. P., LEFOHN, A., COMBA, J. A. L. D., AND SILVA, C. T. 2007. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '07, 97–104.

BUNNELL, M. 2005. Dynamic Ambient Occlusion and Indirect Lighting. In *GPU Gems 2*, 223–233.

CARR, N. A., HALL, J. D., AND HART, J. C. 2003. GPU algorithms for radiosity and subsurface scattering. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 51–59.

DEMERS, J. 2004. Depth Of Field: A Survey Of Techniques. In *GPU Gems 1*, 375–389.

ENGEL, W., 2008. Light Pre-Pass Renderer. `http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html`.

FILION, D., AND MCNAUGHTON, R. 2008. Effects & Techniques in Starcraft 2. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, ACM, New York, NY, USA, 133–164.

GREEN, S. 2004. Real Time Approximations To Subsurface Scattering. In *GPU Gems 1*, 263–278.

HAMMON, E. 2008. Practical Post-Process Depth of Field. In *GPU Gems 3*, 583–605.

HAO, X., AND VARSHNEY, A. 2004. Real-time rendering of translucent meshes. *ACM Trans. Graph. 23*, 2, 120–142.

HAO, X., BABY, T., AND VARSHNEY, A. 2003. Interactive subsurface scattering for translucent meshes. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 75–82.

HARGREAVES, S., AND HARRIS, M., 2004. 6800 Leagues under the sea - Deferred Shading. `http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf`.

HEIRICH, A., AND BAVOIL, L., 2005. Deferred Pixel Shading on the Playstation 3. `http://research.scea.com/ps3_deferred_shading.pdf`.

HUTCHINSON, N., KNIGHT, B., RITCHIE, M., PARRISH, G., AND MOORE, J. 2010. Screen space classification for efficient deferred shading. In *SIGGRAPH '10: ACM SIGGRAPH 2010 Talks*, ACM, New York, NY, USA, 1–1.

JARED, H., AND JIA, Y. 2008. High-Quality Ambient Occlusion. In *GPU Gems 3*, 257–274.

KIRCHER, S., AND LAWRANCE, A. 2009. Inferred lighting: fast dynamic lighting and shadows for opaque and translucent objects. In *Sandbox '09: Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, ACM, New York, NY, USA, 39–45.

KOONCE, R. 2008. Deferred Shading in Tabula Rasa. In *GPU Gems 3*, 429–457.

LORENZON, J., AND CLUA, E. 2009. A novel multithreaded rendering system based on a deferred approach. In *Games and Digital Entertainment (SBGAMES), 2009 VIII Brazilian Symposium on*, 168 –174.

MATTAUSCH, O., BITTNER, J., AND WIMMER, M., 2008. Chc++: Coherent hierarchical culling revisited, Apr.

MITTRING, M. 2007. Finding NextGen: CryEngine 2. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, ACM, New York, NY, USA, 97–121.

MOHAMMADBAGHER, M., KAUTZ, J., HOLZSCHUCH, N., AND SOLER, C. 2010. Screen-space Percentage-Closer Soft Shadows. In *SIGGRAPH '10: ACM SIGGRAPH 2010 Posters*, ACM, New York, NY, USA, 1–1.

PHARR, M., AND GREEN, S. 2004. Ambient Occlusion. In *GPU Gems 1*, 279–292.

PHARR, M., AND HUMPHREYS, G. 2004. In *Physically Based Rendering*.

POLICARPO, F., AND FONSECA, F., 2005. Deferred Shading Tutorial. `http://www710.univ-lyon1.fr/~jciehl/Public/educ/GAMA/2007/Deferred_Shading_Tutorial_SBGAMES2005.pdf`.

RITCHIE, M., MODERN, G., AND MITCHELL, K. 2010. Split second motion blur. In *SIGGRAPH '10: ACM SIGGRAPH 2010 Talks*, ACM, New York, NY, USA, 1–1.

RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, 75–82.

ROSADO, G. 2008. Motion Blur as a Post-Processing Effect. In *GPU Gems 3*, 575–581.

SHISHKOVTSOV, O. 2005. Deferred Shading in S.T.A.L.K.E.R. In *GPU Gems 2*, 143–166.

SOLER, C., HOEL, O., AND ROCHET, F. 2010. A deferred shading pipeline for real-time indirect illumination. In *SIGGRAPH '10: ACM SIGGRAPH 2010 Talks*, ACM, New York, NY, USA, 1–1.

TREAVETT, S. M. F., AND CHEN, M. 2000. Pen-and-ink rendering in volume visualisation. In *VIS '00: Proceedings of the conference on Visualization '00*, IEEE Computer Society Press, Los Alamitos, CA, USA, 203–210.

TULIP, J., BEKKEMA, J., AND NESBITT, K. 2006. Multi-threaded game engine design. In *Proceedings of the 3rd Australasian conference on Interactive entertainment*, Murdoch University, Murdoch University, Australia, Australia, IE '06, 9–14.

VALIENT, M., 2007. Deferred Rendering in Killzone 2. `http://www.develop-conference.com/developconference/downloads/vwsection/Deferred%20Rendering%20in%20Killzone.pdf`.