Chapter 5

Effects & Techniques

Dominic Filion[1]
Rob McNaughton[2]

BLIZZARD ENTERTAINMENT

[1] dfilion@blizzard.com
[2] rmcnaughton@blizzard.com

Figure 1. *A screenshot from* StarCraft II

5.0 Abstract

In this chapter we present the techniques and algorithms used for compelling storytelling in the context of the *StarCraft II*© real-time strategy game. We will go over some of the design goals for the technology used to empower our artists for both in-game and "*story mode*" settings as well as describe how the Blizzard art style influenced the design of the engine. Various aspects of our lighting pipeline will be unveiled, with a strong focus on several techniques making use of deferred buffers for depth, normals, and coloring components. We will show how these deferred buffers were used to implement a variety of effects such as deferred lighting, screen-space ambient occlusion and depth of field effects. Approaches with respect to shadows will also be discussed.

5.1 Introduction

Starcraft II presented unique challenges for development as a second installment in a well-known franchise with the first entry dating from ten years before. During the process of development we had to overcome these and we will share the engineering

choices that were made in the context of the *Starcraft II* graphics engine, and in particular how these choices were made to support the unique Blizzard art style.

5.2 Engine Design Goals

Early on during development, we had several themes in mind that we wanted to drive the development of our graphics engine with:

Scalability First

A major design goal is the scalability of the engine. Blizzard games are well known for their ability to support a range of consumer hardware, including fairly outdated specifications, while always providing a great player experience. This was typically achieved in our previous games by keeping the playing field pretty even between all players, with a minimal set of video options. For *Starcraft II*, we wanted to maximize compatibility with less capable systems to ensure hassle-free game play for as broad a player base as possible. Yet we also wanted to utilize the full potential of any available hardware to ensure the game's looks were competitive. This meant supporting a wide range of hardware, from ATI Radeon™ 9800/NVIDIA GeForce™ FX's to the ATI Radeon™ HD 4800s and NVIDIA GeForce™ G200s, targeting maximum utilization on each different GPU platform.

This scalability translated in a fairly elaborate shader framework system. A full discussion of our shader system could very well encompass a whole chapter by itself so we will not focus on it here, so this is only a short overview.

We only had one or two artists that would actually have any interest in writing their own shaders so early on, instead of focusing on writing elaborate shader prototyping tools for artists we decided to focus our efforts on making the shader framework as flexible and easy to use for programmers as possible, which is somewhat counter to the industry trend at this time.

Thus, writing shader code in our game is generally very close to adding a regular C++ file in our project. Figuratively, we treat the shader code as an external library called from C++, with the shader code being a free form body of code organized structurally as a C++ codebase would. Thus, the concept of shaders can be a loose one in *Starcraft II* – the shader code library defines several entry points that translate to different shaders, but one entry point is free to call into any section of shader code. Thus it would be difficult to talk about how many "individual" shaders *Starcraft II* uses, as it is a single body of code from which more than a thousand shader permutations will generally be derived for a single video options configuration. Making our shader framework system as familiar as possible for programmers has enabled us faster turnaround time when debugging shaders. In our case, when our technical artists have some ideas for new

shaders they will generally prototype it using 3D Studio MAX renders and a programmer will implement an optimized version of the effect, often times adding reusable abstractions to the shader that may not have been apparent to an artist. In all, *Starcraft II* uses about 8,000 unique, non-repeating lines of shader code, divided amongst 70 files. Interestingly enough, we've come to the point where the body of shader code alone has grown larger than the entire codebase for games from the early stages of the games industry.

Stress GPU over CPU

We chose early on to stress the GPU more than the CPU when ramping up quality levels within the game. One of the main reasons for this is that, in *Starcraft II*, you are able to spawn and manage potentially hundreds of the smaller base units such as *zerglings* and *marines*. In large-scale multiplayer games with eight players, this can translate to up to roughly five hundred units on the screen at a single time, at peak. Because the number of units built is largely under the control of the player (as well as due to the choice of the selected race), balancing the engine load such that CPU potential is well utilized in both high-unit count and low-unit unit count situations becomes cumbersome.

Figure 2. *Players can control a very high number of units, thus balancing batch counts and vertex throughputs is key to reliable performance.*

In contrast, GPU loads in our case tend to be much more affected by the pixel shader load, which tends to stay constant for a real-time strategy game because of the generally low overdraw. Although vertex counts for a particular scene can ramp up well above a million in a busy battle scene, this type of vertex throughput is well handled by most modern GPU hardware. This has driven us to push the engine and art styles towards ever increasing pixel-shader level effects whenever possible, minimizing batch counts and using relatively conservative vertex counts.

The *Starcraft II* units are generally viewed from a good distance away and therefore result in a small screen-space footprint during game play. Therefore, increasing the model vertex counts would have given us marginal benefits during actual game play in most cases. For that reason we avoided generating art assets with large polygon counts.

Dual Nature of the Engine

Starcraft II is supported by an engine that in many ways has a split personality; during normal game play we typically render scenes from a relatively far away distance, with high batch counts, and a focus on action rather than details. At the same time, we really wanted to push our storytelling forward with *Starcraft II*, and this is where the game's "*Story Mode*" comes in. In this mode, the player generally sits back to take in the game's rich story, lore and visuals, interacting with other characters through dialogues and watching actions unfold. This is the mode where most of *Starcraft II*'s storytelling happens and it has radically different and often opposing constraints to the in-game mode; story mode generally boasts lower batch counts, close-up shots, and a somewhat more contemplative feel – all things more typical of a first person shooter.

Figure 3. *In-game view vs. "story mode" in Starcraft II.*

We will explore how these different design goals affected our algorithmic choices. We will showcase some of our in-game cinematics and peel off some of the layers of technology behind it, as well as shows examples of the technology in actual gameplay.

5.3 Screen-Based Effects

One of the objectives for *Starcraft II* was to present rich lighting environments in story mode, as well as more lighting interactions within the game itself. Previously in *Warcraft III*, each unit had a hard limit as to the number of lights that could affect it at any given time. This would cause light popping as units moved from one of light influences to another. For that reason, the use of dynamic lighting was fairly minimal. At the same time, using forward rendering approaches, we were quickly presented with the problem of the large increase in the number of draw call batches. One of the stress cases we used was a group of marines, with each marine casting a flickering light that in turn shades the surrounding marines around him. The batch counts in such cases rapidly became unmanageable, and issues were also present with our deeply multi-layered terrain rendering, thus requiring complex terrain slicing and compounding the problem.

The goal to reduce batch counts was the first step that sent us on the road towards using deferred buffers for further treatment at the end of the frame. "Deferred rendering" term (as covered in [CALVER04], [SHISHKOVTSOV05] and in [KOONE07]) often refers to the storage of normals, depth and colors to apply lighting at a later, "deferred" stage. In our case we use it in the broader sense to mean any graphical technique which uses this type of deferral, even in cases where a non-deferred approach would have been viable as well. We'll explore issues and commonalities with all things deferred.

In practice, we've found many pros and cons for using deferred computations with the storage of relevant information, such as depth values, normals, etc. Although storage of this information consumes memory and bandwidth, as well as the extra cost of re-sampling the buffers at the processing stage, it generally helps greatly with scalability by ensuring the cost curve of effects tend to stay linear instead of exponential. When using deferred buffers, adding more layers of effects generally results in a linear, fixed cost per frame for additional full-screen post-processing passes regardless of the number of models on screen, whereas the same effect with forward rendering exhibits an exponential cost behavior. This leads us unto our goal of maximizing resource usage regardless of whether there five or five hundred units on the screen.

Keeping batch counts low is paramount for any RTS game. We were thus naturally drawn away from trying to fill up the deferred buffers in a separate pass and instead relied on multiple render targets (MRTs) to fill the deferred buffers during the main rendering. Thus, in the main rendering pass, several MRTs will be bound and filled with their respective information. Much of the mainstream hardware is limited to four

render targets bound at once, so right now it makes sense to pick deferred component that will fit within the sixteen individual channels this provides for.

For *Starcraft II*, any opaque model rendered will store the following to multiple render targets bound in its main render pass:

- Color components not affected by local lighting, such as emissive, environment maps and forward-lit color components;
- Depth;
- Per-pixel normal;
- Ambient occlusion term, if using static ambient occlusion. Baked ambient occlusion textures are ignored if screen-space ambient occlusion is enabled;
- Unlit diffuse material color;
- Unlit specular material color.

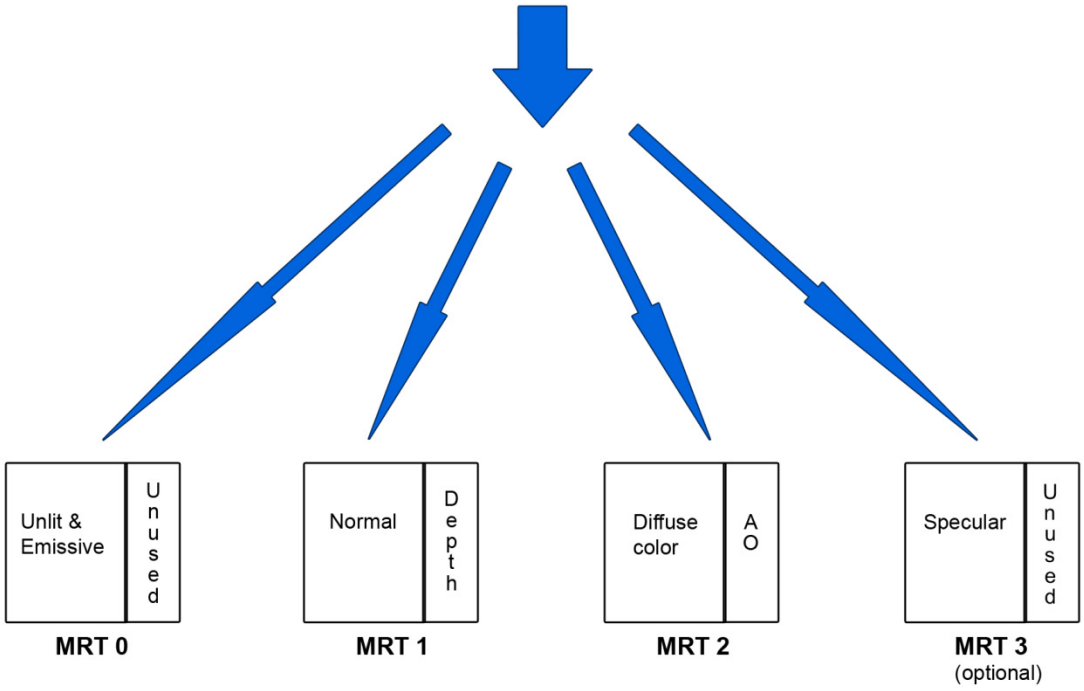

Figure 4. *MRT setup.*

As a quick reminder, simultaneously bound render targets under DirectX® 9 need to be of the same width and height, and in most cases, the same bit depth. Some, but not all, hardware supports independent color write control on each MRT which should be used whenever possible to minimize wasted bandwidth for unused components. For example, the alpha component is not used on all of these buffers.

We are making heavy use of HDR ([RWPD05])in *Starcraft II*, and thus all these buffers will normally be four-channel 16-bit floating point formats. Using higher precision format helps to sidestep accuracy issues and minimizes pixel shader instructions for

decoding the buffers. The restriction that all buffers should be of the same bit depth unfortunately forces the use of much wider buffers than we actually need for a many of these buffers, pushing the output bandwidth to 24 bytes per pixel. We've found however that even with this very heavy bandwidth cost the freedom this allows us in the placement of our lighting was well worth the expense.

In the majority of cases, all objects outputting to these buffers are opaque, with some notable exceptions; we will come back to the problem of handling rendering transparent objects later.

Our terrain is multi-layered, in which case the normals, diffuse and specular colors are blended in these buffers, while leaving the depth channel intact – only the bottom terrain layer writes to the depth buffer.

Rendering to the MRTs provides us with per-pixel values that can be used for a variety of effects. In general, we use:

- *Depth values* for lighting, fog volumes, dynamic ambient occlusion, and smart displacement, depth of field, projections, edge detection and thickness measurement.
- *Normals* for dynamic ambient occlusion
- *Diffuse and specular* for lighting

5.4 Deferred Lighting

Deferred lighting in *Starcraft II* is used for local lights only: point and spot lights with a defined extent in space. Global directional lights are forward rendered normally; because these lights have no extents and cover all models, there is little benefit in using deferred rendering methods on them, and it would actually be slower to resample the deferred buffers again for the entire screen.

The computation results for deferred rendering vs. forward rendering equations are equivalent, yet the deferred form is more efficient for complex lighting due to the tighter light coverage offered by rendering the light shapes as a post-process. For the most part, the new equation simply moves terms from one stage of the rendering pipeline to a later stage, with the notable exception of the pixel's view space position, which is more efficiently reconstructed from the depth information.

Pixel Position Reconstruction

Pixel shader 3.0 offers the VPOS semantic which provides the pixel shader with the x and y coordinates of the pixel being rendered on the screen. These coordinates can be normalized to the [-1..1] range based on screen width and height to provide a normalized eye-to-pixel vector. Multiplying by the depth will provide us with the pixel's

view space position. For pixel shader 2.0, a slightly slower version is used where the vertex shader feeds to the pixel shader a homogeneous equivalent to VPOS, from which the w component must be divided in the pixel shader.

```
float3 vViewPos.xy = INTERPOLANT VPOS * half2( 2.0f, -2.0f ) + half2( -1.0f, 1.0f ) ) *
                     0.5 * p vCameraNearSize * p vRecipRenderTargetSize;

vViewPos.zw  = half2( 1.0f, 1.0f );
vViewPos.xyz = vViewPos.xyz * fSampledDepth;

float3 vWorldPos = mul( p_mInvViewTransform, vViewPos ).xyz;
```

Listing 1. *Pixel position reconstruction*

Stenciling, Early-Z and Early-Stencil

In our case, usage of early stencil out provided substantial speed improvements for our scenes. Stenciling allows us to discard pixels that are covered by the light shapes but whose depth is too far behind the light to be affected by it.

By the same token, pixels that are in front of the light's influence will automatically benefit from early-z out as the light shape pixels will be discarded early on.

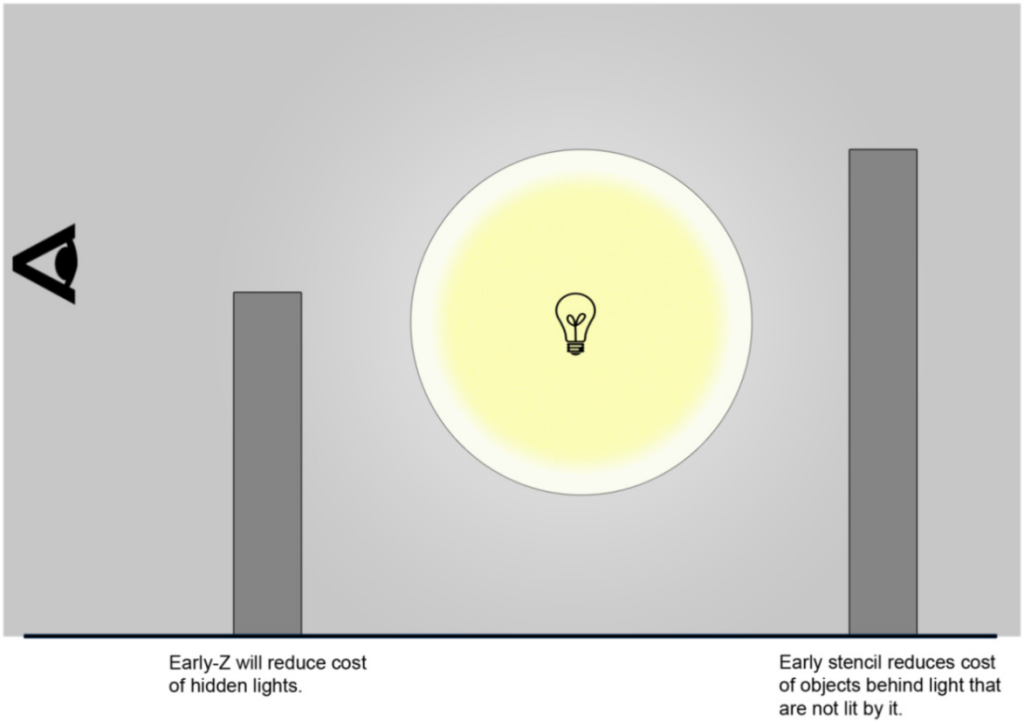

Figure 5. *Deferred rendering interaction with z-buffer and stencil buffer.*

To benefit from early stencil, the light shapes must be rendered again with color writes off before its normal lighting pass. The stencil operation is slightly different depending on whether the viewing camera sits inside the light shape – this has to be tested on the CPU.

If the camera is outside the light shape, effectively we want to only pass through pixels where only one side of the light shape is visible (which will always be the front facing parts); this implies the pixel of the surface being lit is hiding the back facing part of the light shape, aka the surface pixel is inside the light shape. This can be achieved by clearing the stencil and passing any pixels that show a front face but no back face for the light shape.

If the camera is inside the light shape, then only the back side of the light shape is visible. In this case we want to color any of the light shape's back facing pixels that failed the z-test, which implies there is a lit surface inside the light shape.

Deferred Lighting in Action

The low batch counts of the deferred system allowed us to truly refine the lighting in our cinematic scenes. Here we see a shot of one of our scenes, which consists of roughly x (fifty) dynamic lights in the same room, from small Christmas lights to larger lighting coming from the TV screen.

Figure 6. *Deferred lighting allows us to use complex lighting environments.*

5.5 Screen-Space Ambient Occlusion

SSAO Basics

Although our initial interest in screen space ambient occlusion was sparked by the excellent results that we observed in such games as Crysis©, we arrived at our solution independently. In a nutshell, the main idea behind screen space ambient occlusion is to approximate the occlusion function at points on visible surfaces by sampling the depth of neighboring pixels in screen space. The resulting solution will be missing occlusion cues from objects that are currently hidden on the screen, but since ambient occlusion tends to be a low frequency phenomenon, the approximation is generally quite convincing.

Figure 7. *Scene with lighting information only; softer shading cues are accomplished through SSAO.*

SSAO requires depth to be stored in a prior step; for *Starcraft II* this is the same depth buffer that we use for deferred lighting. The ambient occlusion term that will be produced by the SSAO is stored in the alpha channel of our diffuse deferred render buffer, which is then used to modulate lighting from certain global and local lights.

At any visible point on a surface on the screen, multiple samples (8 to 32) are taken from neighboring points in the scene. These samples are offset in 3D space from the current point being computed; they are then projected back to screen space to sample the depth at the sample location.

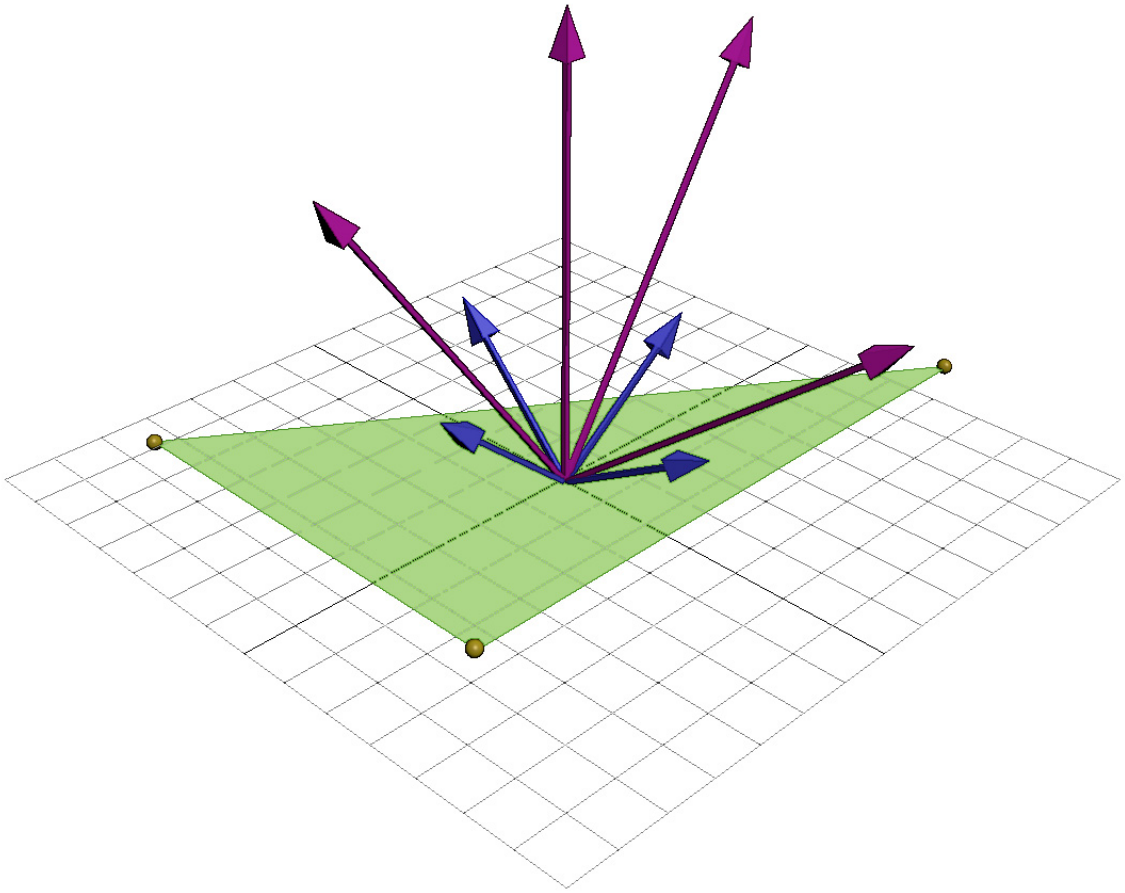

Figure 8. *Overview of SSAO sampling process.*

The objective is to check if the depth sampled at the point is closer or further away than the depth of the sample point itself. If the depth sampled is closer, than there is a surface that is covering the sample point. In general this means there is another surface nearby that is covering the area in the vicinity around the current pixel, and some amount of occlusion should be occurring. It is the average of these samples that will determine the total occlusion value for the pixel.

The depth test of the sample point with its sampled depth is not simply a Boolean operation. If the occluding surface is very close to the pixel being occluded, it will occlude a lot more than it is further away; and in fact beyond a certain threshold there needs to be no occlusion at all as we don't want surface far away from the surface to occlude it. Thus we need some type *occlusion function* to map the relationship between

the depth delta between the sample point and its sampled depth, and how much occlusion occurs.

If the aim is to be physically correct, then the occlusion function should be quadratic. In our case we were more concerned about being able to let our artists adjust the occlusion function, and thus the occlusion function can be arbitrary. The occlusion functions can be any function that adheres to these criteria:

- Negative depth deltas should give zero occlusion (the occluding surface is behind the sample point);
- Smaller depth deltas should give higher occlusion values;
- The occlusion value needs to fall to zero again beyond a certain depth delta value.

For our implementation we simply chose a linearly stepped function that is entirely controlled by the artist. There is a *full occlusion* threshold where every positive depth delta smaller than this value gets complete occlusion of one, and a *no occlusion* threshold beyond which no occlusion occurs. Depth deltas between two extremes fall off linearly from one to zero, and the value is exponentially raised to a specified *occlusion power* value. If a more complex occlusion function is required, it can be pre-computed in a small 1D texture to be looked up on demand.

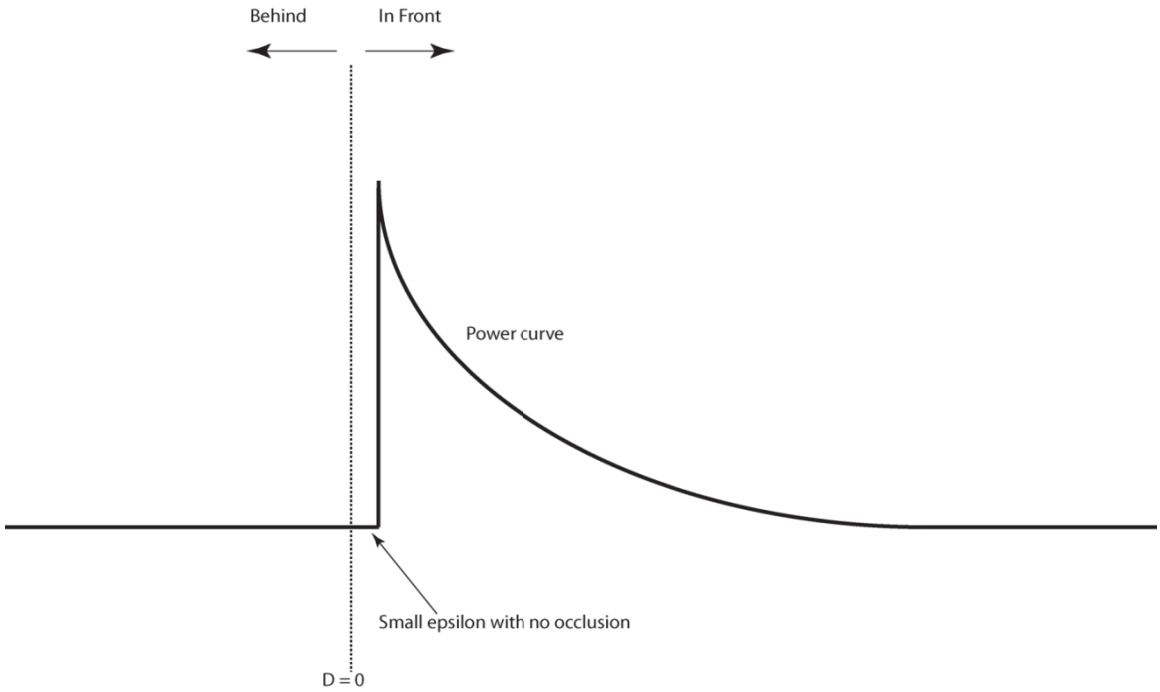

Figure 9. *Occlusion function.*

To recap, the SSAO generation process is:

- Compute the view space position of a pixel;

- Add (8 to 32) offset vectors to this position;
- Remap these offset vectors to where they are in screen space;
- Compare the depth of each offset vector with the depth at the point where the offset is;
- Each offset contributes occlusion if the sampled depth on screen is behind the depth of the offset vector. Occlusion factor is based on the difference between the sampled depth and the depth of the offset vector.

All in the Details: SSAO Artifacts

The basics of SSAO are fairly straightforward but there several alterations that need to be made for it to produce acceptable results.

Sampling Randomization

To smooth out the results of the SSAO lookups, the offset vectors must be randomized thoroughly. One good approach, as given in (6) is to generate a 2D texture of random normal vectors and lookup this texture in screen space, thus retrieving a unique random vector per pixel on the screen. More than one random vector must be generated per pixel however: these are generated by passing a set of offset vectors in the pixel shader constant registers and reflecting these vectors through the sampled random vector, thus generating a semi-random set of vectors at each pixel. The set of vectors passed in as registers is not normalized – having varying lengths helps to smooth out the noise pattern. In our case we randomize the length of the offset vectors from 0.5 to 1, avoiding samples clustered too close to the source point. The lengths of these vectors are scaled by an artist-controlled parameter that determines the size of the sampling area.

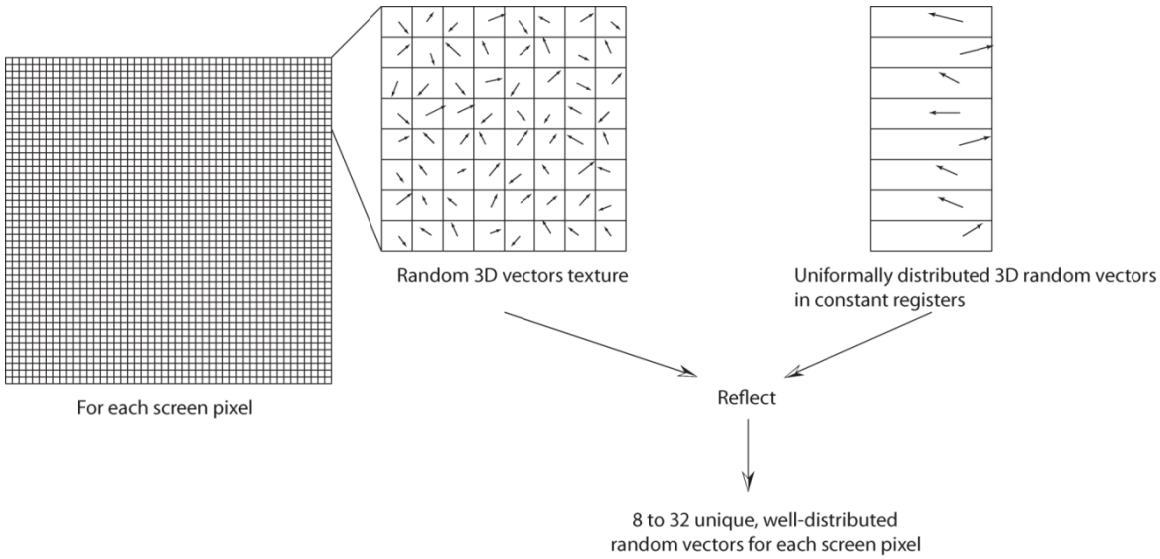

Figure 10. *Randomized sampling process.*

Figure 11. *SSAO term after random sampling applied. Applying blur passes will further reduce the noise to achieve the final look.*

Blurs

The previous step helps to break up the noise pattern, producing finer grained pattern that is less objectionable. With wider sampling areas however, more processing such as blurs become necessary. The ambient occlusion results are low-frequency, and we've found losing some of the high-frequency detail due to blurring was not an issue.

The ambient occlusion must not bleed through edges to objects that are physically separate within the scene, and thus a form of smart Gaussian blur is used. This blur samples the nearby pixels as a regular Gaussian blur shader would, yet the normal and depth for each of the Gaussian samples is sampled as well (encoding the normal and depth in the same render targets presents significant advantages here). If either the depth from Gaussian sample differs from the center tap by more than a certain threshold, or the dot product of the Gaussian sample and the center tap normal is less than a certain threshold value, then the Gaussian weight is reduced to zero. The sum of the Gaussian samples is then renormalized to account for the missing samples.

Several blur passes can thus be applied to the ambient occlusion output to completely eliminate the grain pattern.

Self-occlusion

The output from this produces convincing results, but has the issue that in general no area is ever fully unconcluded due to the fact that the random offset vectors from each point will penetrate through the surface of the object itself, and the object becomes self-occluding at all times.

One possible solution around this is to generate the offset vectors around a hemisphere centered around the normal at that point on the screen (which implies a deferred normal buffer has to be generated, another strong point for reusing output from deferred rendering). Transforming each offset vector by a matrix can be expensive however, and one compromise we've used is to instead perform a dot product between the offset vector and the normal vector at that point, and to flip the offset vector is the dot product is negative. This is a cheaper way to solve the problem at the expensive of possibly making the noise pattern more predictable.

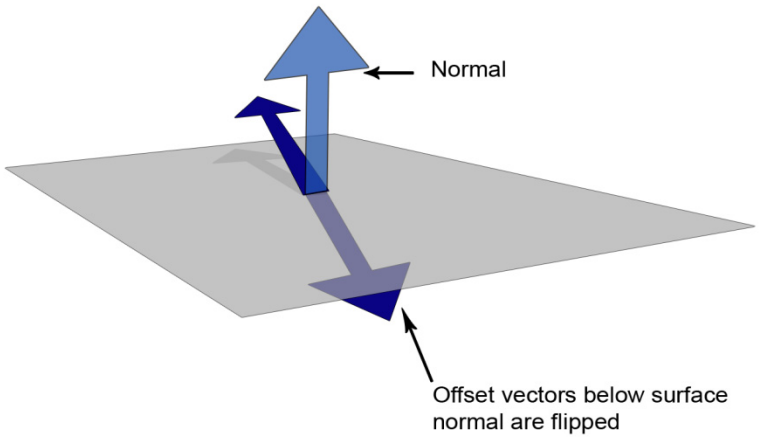

Figure 12. *Handling self-occlusion.*

Edge Cases

The offset vectors are in view space, not screen space, and thus in close-up camera shots these vectors will be longer so that the SSAO sampling area remains constant within the scene. This can mean more noise for close-ups, but also presents problems when samples end up looking up outside the screen. There is no straightforward way to deal with the edge case, as the depth information outside the screen is not present:

rendering a wider area than what is seen in screen could help improve this, but is not a robust solution that would work in all cases. The less objectionable way to deal with this is to ensure that samples outside the screen return a large depth value, ensuring they would never occlude any neighboring pixels. This can achieved through the "border color" texture wrapping state.

To prevent unacceptable breakdown of the SSAO quality in extreme close-ups, we've found it necessary to impose an SSAO screen-space sampling area limiter. In effect, if the camera is very close to an object and the SSAO samples end up being too wide, the SSAO area consistency constraint is violated so that the noise pattern doesn't become too noticeable. Another alternative would to simply vary the number of samples based on the sampling area size, but this can introduce wide frame rate swings that we have been eager to avoid.

SSAO Performance

SSAO can give a significant payback in terms of mood and visual quality of the image, but it can be quite an expensive effect. The main bottleneck of the algorithm is the sampling itself: the random nature of the sampling, which is necessary to minimize noise, wreaks havoc with the GPU's texture cache system and can become a problem if not managed. The performance of the texture cache will also be very dependent on the sampling area size, with wider areas straining the cache more and yielding poorer performance. Our artists quickly got in the habit of using SSAO to achieve a faked global illumination look that suited their purposes: this required more samples and wider sampling areas, so extensive optimization become necessary for us.

One method to bring SSAO to an acceptable performance level relies on the fact that ambient-occlusion is a low-frequency phenomenon. Thus we've found there is generally no need for the depth buffer sampled by the SSAO algorithm to be at full screen resolution. The initial depth buffer can be generated at screen resolution, since we reuse the depth information for other effects within the game and it has to fit the size of the other MRTs, but it is thereafter down sampled to a smaller depth buffer that is a quarter size of the original on each side. The down sampling itself does have some cost but the payback in improved texture cache afterwards is very significant.

SSAO and Global Illumination

Wide-area SSAO vs. Outline Enhancement

One thing we've quickly discovered is that our cinematic artists loved the quasi-global illumination feel of the output. If the sampling area of the SSAO is wide enough, the look of the scene changes from darkness in nooks and crannies to a softer, ambient feel.

The SSAO implementation was thus pulled by the art direction into two somewhat conflicting directions: on the one hand, the need for tighter, high-contrast occluded zones in deeper recesses, and on the other hand, the desire for the larger, softer ambient look of the wide area sampling.

In our case, we resolved this conundrum by splitting the SSAO samples between two different sets of SSAO parameters: one quarter of our samples are concentrated in a small area with a rapidly increasing occlusion function, while the remainder uses a wide sampling area with a gentler function slope. The two sets are averaged independently and the final result uses the value from the set which produces the most (darkest) occlusion.

5.6 Depth of Field

Figure 13. *Depth of field as used in Starcraft II for a cinematic feel.*

Because deferred rendering had to use generate a normal and a depth buffer every frame, we were naturally drawn to leverage this as much as possible. One of the post-processing effects crucial for many of our story mode scene has been depth of field. There has been a fair amount of interest in real-time depth of field rendering, as covered in [ScheuermannTatarchuk04] and [Demers05], for example. For storytelling we will often focus the camera on a specific object to attract the attention of the viewer. Here is a review of the problems we faced and how they were solved.

Circle of Confusion

The ingredient needed to perform the depth of field, which is common for real-time depth of field algorithm, is to compute a blur factor, or circle of confusion (CoC) for each pixel on the screen. We only are loosely to the definition of the circle of confusion, which would be the radius of the circle over which pixels, or rays, are blurred – the only point of importance is that higher levels of CoC needs to map to a blurrier image in some kind of predictable gradient.

Thus, the circle of confusion is art-driven instead of physics-driven. Artists specify a reference point that is a certain distance in front of the viewing camera. From this point, a *full focused* distance is specified – all depths whose distance from the reference point, either in front or behind the reference point, is less than this distance are sharp. Beyond this distance, the image progressively gets blurrier until it reaches the *fully unfocused* distance where the blurriness is maximal.

Thus computing the circle of confusion comes down to sampling from the deferred depth buffer and going through the following equation:

$$saturate\left(\frac{DofAmount \times max(0, Depth - FocalDepth - NoBlurRange)}{MaxBlurRange - NoBlurRange}\right)$$

Equation 1. CoC as a function of depth.

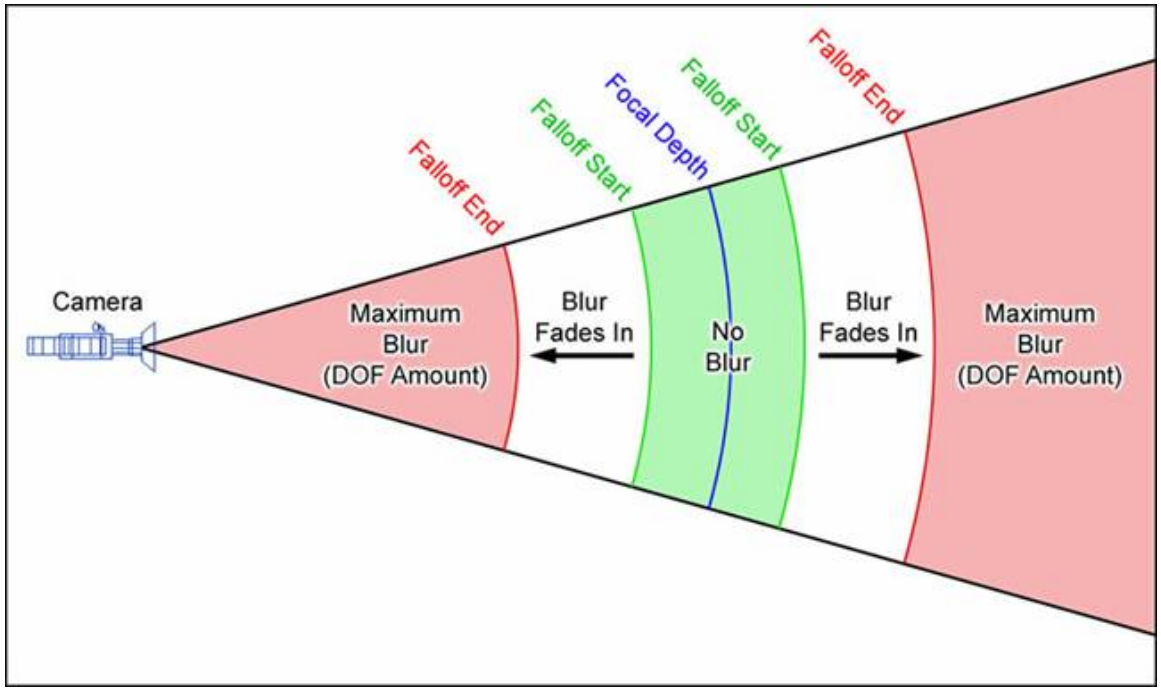

Figure 14. *Depth of field regions.*

Blurring

The circle of confusion value, for each pixel, can then be used to gradually blur the image. The blur needs to be gradual as the CoC value increases, and not present under-sampling artifacts.

Simply varying the width of the Gaussian filter kernel to simulate different blur levels doesn't work too well; larger blur factors require more blur samples to look as good. This is achievable by varying the number of samples per pixel, which unfortunately requires use of dynamic branching in the pixel shader, whose market penetration wasn't as high as we needed it to be.

Thus a different approach is to generate pre-blurred images at fixed levels and linearly interpolate the outputs from the two closest pre-blurred sample points. In practice, this means blurring between four different reference images: the sharp image, two other screen-sized images of increasing blurriness, and a maximal blur image with each side a quarter of the original.

These four images can be generated, and the depth of field image effects can then compute the CoC factor and use it to linearly blend between two of the images.

Edge Cases

The two previous steps work well to create the effect of blurriness in different area of the screen based on the pixel depth, but edges pose problems that require special care. To be convincing, depth of field needs to exhibit the following behaviors:

- Out-of-focus foreground objects should bleed unto objects behind them, whether sharp or unfocused;
- Conversely, bleeding of unfocused background objects unto foreground objects should be avoided – clearly visible, focused edges need to stay sharp.

However, the blurring process occurs by *looking up* samples from neighboring pixel and doing a weighted, which requires us to restate these statements in a way that maps better to how the algorithm is actually implemented, as follows:

- *Avoiding sharp halos:* Sharp pixels should not contribute to the weighted sum of any neighboring pixels or they will create halos as well - partially unfocused pixels should contribute more to the sum of neighboring pixels than more focused ones.
- *Bleeding of blurry objects over background:* the blurriness of a single pixel does not depend on the circle of confusion of that pixel alone; a nearby blurry pixel with a large circle of confusion would have the effect of making the current pixel blurry;
- *Depth ordering of blurriness:* Pixels in the background should not contribute to the weighted sum of pixels in the foreground;

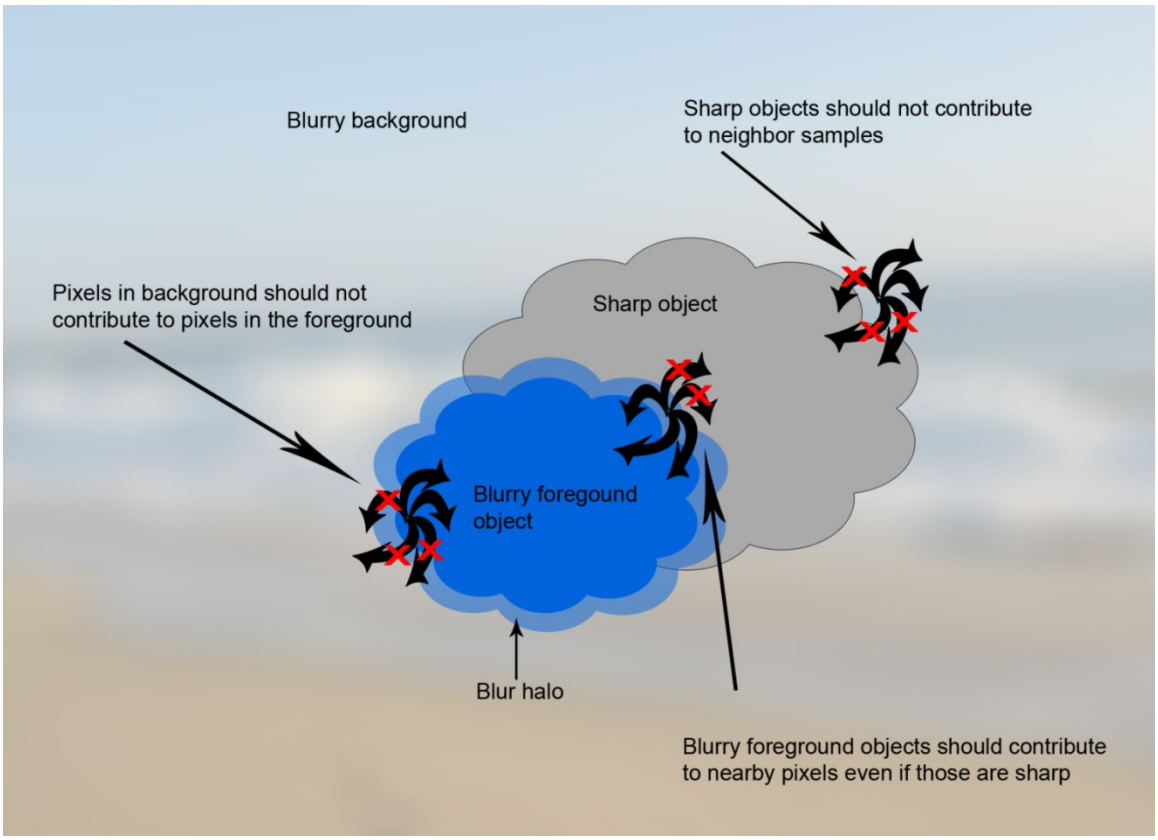

Figure 15. *Depth of field rules.*

For the first constraint, we must weigh the Gaussian samples by the corresponding CoC factors for each pixel. The sampled CoCs are renormalized so that the sum of all sampled CoCs adds up to one and each sampled CoC is multiplied by the corresponding sample. Thus, a sharp pixel will contribute less (or not at all) to neighboring pixels that are blurry. Although the effect may not be correct in a strict sense, the most important goal here is to eliminate any changes in color for blurry pixels when completely focused, neighboring pixels are changing.

The second constraint can be solved by blurring the circle of confusion factors in an image a quarter size per side, ensuring that any blurry pixels will cause neighboring pixels to become blurry as well.

Blurring all CoCs will violate the third constraint however, causing background objects to blur over foreground ones – we need to conserve some sense of depth ordering of the blur layers. We can achieve this through the following process:

- Downscale and blur the *depth map* in an image a quarter size per side;
- Sample both the blurred and non-blurred depth maps;
- If the blurred depth is smaller (closer) than the non-blurred depth, use the CoC from the *blurred CoC map;*

- If the blurred depth is larger (further away) than the non-blurred depth, compute and use the CoC for the current pixel only (no CoC blurring).

This effectively compares the Gaussian-weighted average of the cluster of pixels around the current pixel and compares it with depth at that pixel only. If this average is further away than the current pixel, then other pixels in that area tend to be behind the current pixel and do not overlap it. Otherwise they are in front, and their CoC is expected to affect the current pixel's CoC (blur over it).

Putting It All Together

Putting all the constraints in place, we can put everything together using the following process:

- Perform a full-screen pass to **compute the circle of confusion** for each pixel in the *source* image and store the result in the alpha channel of a downscaled *CoC* image buffer a quarter of the size on each side;
- **Generate the medium blur image** by applying a RGB Gaussian blur with each sample weighted by the CoC on the *source* image;
- **Generate the max blur image** by downscaling the RGB of the source image into an image buffer a quarter of the size on each side – the CoC and large blur buffers can be the same since they use different channels;
- **Blur the max blur image** with the RGB samples weighted by the downscaled CoC. The alpha channel, which contains the CoC, also gets blurred here, but its samples are not weighted by itself.
- **Downscale and blur the depth map** into a *downscaled depth image* – it should be noted in our case we reuse the downscaled depth for our SSAO pass as well (but do not blur depth for the SSAO);
- Then apply the final depth of field shader, binding the source image, medium and large blur/blurred CoC image, the non-blurred depth map and the downscaled depth image to the shader. The depth of field shader:
    - Computes the *small* blur value directly in the shader using a small sample of four neighbor pixels;
    - Computes the CoC for that pixel (the downscaled CoC would not match);
    - Samples the non-blurred and blurred depth to compare them – use computed CoC if blurred depth is further away than non-blurred depth, otherwise use CoC value samples from blurred CoC image;
    - Calculate contribution from each of the possible blur images: source, computed small blur color, medium and large blur images based on the CoC factor from zero to one;
    - Sums the contribution of the small, medium and large blurs
    - Output the alpha to include the contribution of the source (no blur) image.

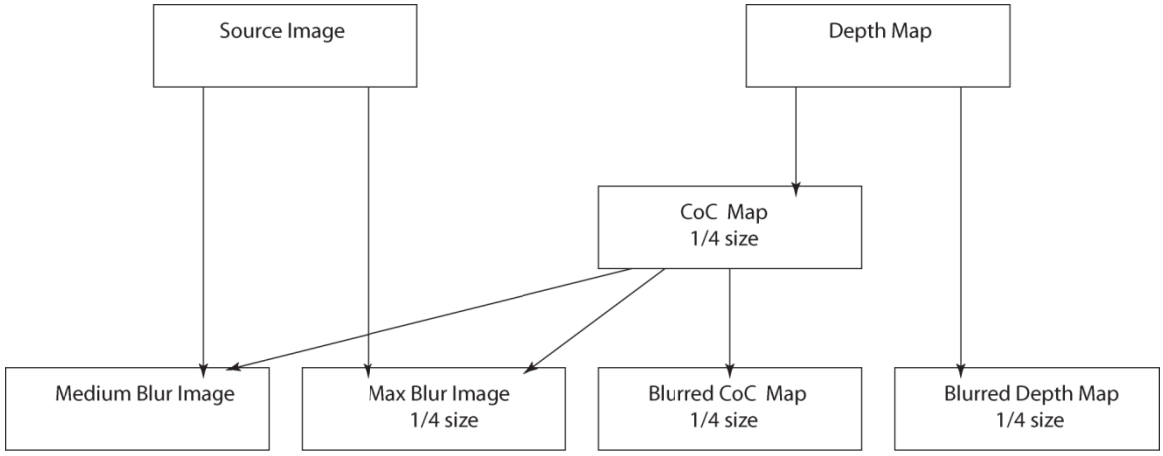

Figure 15. *Depth of field texture inputs & outputs. In the implementation some components reuse the same texture, but they're shown separated here for clarity.*

```
float       p_fDOFAmount;          // Depth of field amount.
float       p_fFocusDistance;      // Distance in focus.
float       p_fFullFocusRange;     // Range inside which everything is in focus.
float       p_fNoFocusRange;       // Range at which everything is fully blurred.
sampler2D   p_sMediumBlurMap;      // Medium blur map for depth of field.
sampler2D   p_sLargeBlurMap;       // Large blur map for depth of field.
sampler2D   p_sCoCMap;             // COC map.
sampler2D   p_sDownscaledDepth;

half ComputeCOC( float depth ) {
       return saturate( p_fDOFAmount * max( 0.0f, abs( depth - p_fFocusDistance ) -
       p_fFullFocusRange ) / ( p_fNoFocusRange - p_fFullFocusRange ) );
}

half4 Tex2DOffset( sampler2D s, half2 vUV, half2 vOffset ) {
    return tex2D( s, vUV + vOffset *
                  half2( 1.0f / p_vSrcSize.x, 1.0f / p_vSrcSize.y ) );}

half3 GetSmallBlurSample( sampler2D s, half2 vUV ) {
    half3 cSum;
    const half weight = 4.0 / 17;
    cSum = 0;
    cSum += weight * Tex2DOffset( s, vUV, half2( 0.5f, -1.5f ) ).rgb;
    cSum += weight * Tex2DOffset( s, vUV, half2( -1.5f, -0.5f ) ).rgb;
    cSum += weight * Tex2DOffset( s, vUV, half2( -0.5f, 1.5f ) ).rgb;
    cSum += weight * Tex2DOffset( s, vUV, half2( 1.5f, 0.5f ) ).rgb;
    return cSum;
}


// Depth of field mode.
half4 PostProcessDOF( VertexTransport vertOut ) {
    float4 vNormalDepth = tex2D( p_sNormalDepthMap, INTERPOLANT_UV.xy );
    float vDownscaledDepth = tex2D( p_sDownscaledDepth, INTERPOLANT_UV.xy ).a;
    half fUnblurredCOC  = ComputeCOC( PIXEL_DEPTH );

    half fCoC = tex2D( p_sLargeBlurMap, INTERPOLANT_UV.xy ).a;

    // If object is sharp but downscaled depth is behind, then stay sharp
    if ( vDownscaledDepth > vNormalDepth.a )
      fCoC = fUnblurredCOC;
```

```
    half d0 = 0.50f;
    half d1 = 0.25f;
    half d2 = 0.25f;
    half4 weights = saturate( fCoC * half4( -1 / d0, -1 / d1, -1 / d2, 1 / d2 ) +
                              half4( 1, ( 1 - d2 ) / d1, 1 / d2, ( d2 - 1 ) / d2 ) );
    weights.yz = min( weights.yz, 1 - weights.xy );

    half3 cSmall    = GetSmallBlurSample( p sSrcMap, INTERPOLANT UV.xy );
    half3 cMed      = tex2D( p sMediumBlurMap, INTERPOLANT UV.xy ).rgb;
    half3 cLarge    = tex2D( p sLargeBlurMap, INTERPOLANT UV.xy ).rgb;

    half3 cColor = weights.y * cSmall + weights.z * cMed + weights.w * cLarge;
    half fAlpha = dot( weights.yzw, half3( 16.0f / 17.0f, 1.0f, 1.0f ) );
    return half4( cColor, fAlpha );
};
```

Listing 2. *Depth of field shader.*

5.7 Dealing with Transparent Object Rendering

It's worthwhile to mention some of the issues with transparency, particularly with respect to the fact that deferred rendering typically do not support transparency very well, or at all. It should be noted that the challenges with transparent objects are not unique to deferred lighting, and in fact if one is striving for absolute correctness, transparency creates problems throughout the entire engine pipeline.

As is typical for any kind of deferred rendering techniques, transparencies are not taken into account by the deferred renderer. We've found however that lit transparencies don't contribute that significantly to the look of the scene as they will pick up the shade of the objects behind them. We do tag some transparent objects selectively as being affected by lighting, in which case these are forward rendered with a multi-pass method. We've been careful to avoid hard cap limits of any sorts on the engine whenever we can, and a multi-pass approach proved to be more scalable than a single-pass one.

Another advantage of the multi-pass lighting approach is that there is no need for more than a single shadow map buffer for the local lighting; the shadow map for each light can be applied one at a time and applied in succession. This proved important because the translucent shadow technology we've designed already needs a second shadow map to render the effect.

We've found that in our environments transparencies are not present in an amount sufficient to create very drastic lighting changes, except for a few specific pieces that we've tagged manually for forward rendering.

So, in light of this, we didn't strive to find clever solutions to fix problems with transparencies with respect to deferred lighting. Other subsystems however, can create

artifacts that are way more noticeable if absolutely no support for transparencies is present.

Depth of field, for instance, has very disturbing artifacts if depth is not taken into account for key transparencies. In this shot, we had a translucent egg that was showing as very crisp along its unfocused neighbors. SSAO also has issues with transparencies, especially for not-quite-opaque objects like bottles that effectively lose their ambient occlusion.

For some of these cases, using a simple layered system worked well enough for us. The depth map is first created from the opaque objects, opaque objects rendered, and depth-dependent post-processing effects subsequently applied on them. Transparent objects are then rendered as they would be normally from back to front, and key transparencies are allowed do a pre-pass where they will output their normals and depth to the deferred buffers. This effectively destroys the depth and normals information for the opaque objects behind these transparencies, but since all post-processing has been applied to these objects already, that information is no longer needed.

After the pre-pass, one more pass is used to update the ambient occlusion deferred buffer. The transparency itself is then rendered, and then another depth of field pass is done specifically on area covered by the transparency.

This certainly makes rendering that particular transparency much more expensive, and we only enable this option for key transparencies where maximum consistency is needed. In simpler cases, we will cheat by allowing the transparency to emit it depth in the depth of field pass (for very opaque objects) or simply work with the art to minimize the artifacts.

5.8 Translucent Shadows

Figure 16. *Translucent shadows allow objects such as smoke and explosion to cast shadows.*

Shadow maps are one of the earlier examples of successfully using screen-space information to resolve shadowing problems that otherwise are much more difficult to deal with in world space. In this section, we will show how extending the shadow map's per-pixel information with some extra channels of information can be used to easily augment shadow maps with translucent shadow support.

The shadow map algorithm is extended with a second shadow map that will hold the information for translucent shadows only; the regular shadow map will still contain the information for opaque shadows. In addition, a color buffer to hold the color of the translucent shadows. However, on most hardware a color buffer of the same size of the shadow map needs to be bound whenever the shadow map is rendered, as most hardware does not support having a null color render target. In most games this color buffer is simply set to the lowest bit depth available to minimize the waste of memory, and the color buffer is otherwise not used. We are thus taking advantage of this "free" color buffer.

In the first pass, the opaque shadow map is filled up as it normally would be, rendering only opaque objects. Then, transparent shadow objects are rendered in our second shadow map for transparent objects with z depth write on, no alpha testing and a regular less-equal z-test. This will effectively record the depth of the closest transparent object to the light.

The transparent shadow color buffer is first cleared to white and is then filled by rendering the transparent shadow objects, from front to back, with the render states thay would normally be used in the regular rendering, no depth write and the less-equal z-test. In this stage we will be using the **opaque** shadow map as the z-buffer when filling the color buffer, ensuring no colors are written for translucencies hidden by opaque objects. The front to back ordering is necessary because we are treating these transparencies as light *filters*. The light hits the front transparencies, and gets filtered as it hits each transparent layer in succession.

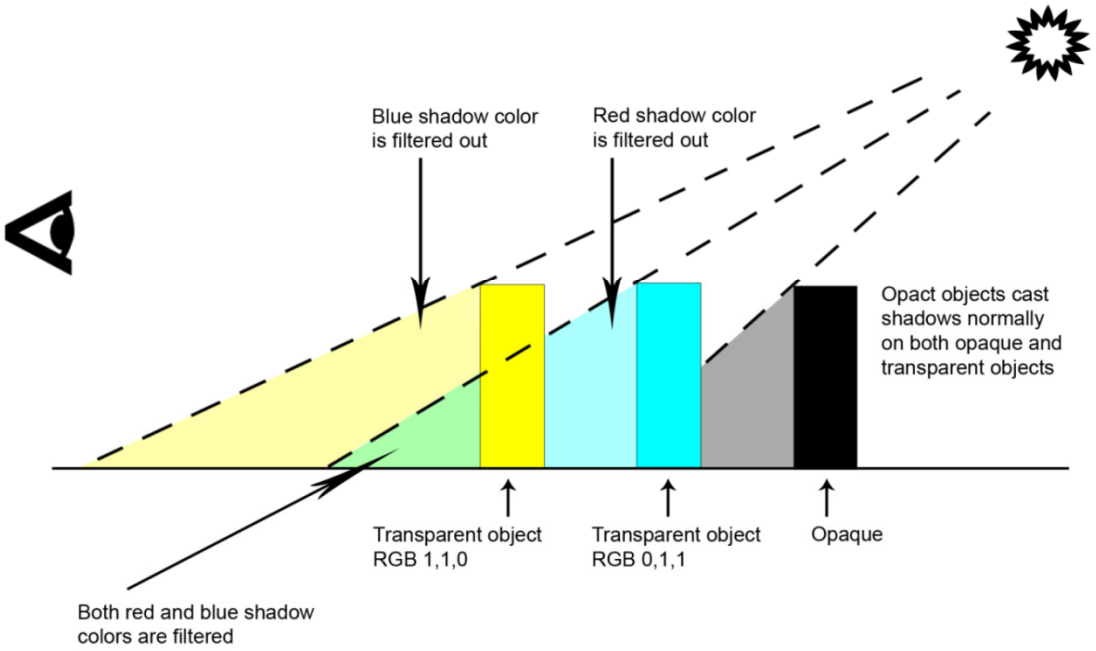

Figure 17. *Light filtering process.*

The rendering of the actual translucent shadows then proceeds as follows:

- Perform shadow test with opaque shadow map and translucencies shadow map
- If we failed the translucent shadow map test, modulate by the color in the transparent shadow color map
- Modulate by the result of the opaque shadow map.

This effectively gives us the following desired behavior:

- Passing both tests doesn't color the light;
- Passing the opaque test but failing the translucent test will always color the light by the translucent shadow color;
- Failing the opaque test will always remove the light contribution.

5.9 Conclusions

Storage of per-pixel information in off-screen buffers opens up a wide array of possibilities for screen-based effects. Screen-space data sets, although incomplete, are straightforward to work with and can help to unify and simply the rendering pipeline. We've shown here a few ways that we record and use these screen-space data sets in Starcraft II to implement effects such as deferred rendering, screen-space ambient occlusion, depth of field, and translucent shadow maps. The resulting per-pixel techniques are easy to scale and manage and tend to be easier to fit in a constant performance footprint that is less dependent on scene complexity.

5.10 References

[CALVER04] CALVER, D. 2004. Deferred Lighting on PS 3.0 with High Dynamic Range, *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, Engel, W. (Editor), Charles River Media, Cambridge, MA, November 2004

[SHISHKOVTSOV05], SHISHKOVTSOV, O. 2005. Deferred Shading in S.T.A.L.K.E.R., *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems),* Pharr, M., Fernando, R. (Editors), Addison-Wesley, March 2005.

[KOONE07], KOONE, R. 2007. Deferred Shading in *Tabula Rasa*, GPU Gems 3, Nguyen, H. (Editor), Addison-Wesley, August 2007

[RWPD05] REINHARD, E., WARD, G., PATTANAIK, S. AND DEBEVEC, P. 2005. High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting, Morgan Kaufmann; Har/Dvdr edition, August 2005

[MITTRING07] MITTRING, M. 2007. Finding Next Gen – *CryEngine 2.0*, Chapter 8, SIGGRAPH 2007 Course 28 – Advanced Real-Time Rendering in 3D Graphics and Games, Siggraph 2007, San Diego, CA, August 2007.

[SCHEUERMANNTATARCHUK04] SCHEUERMANN, T. AND TATARCHUK, N. 2004. Improved Depth of Field Rendering, *ShaderX3: Advanced Rendering with DirectX and OpenGL (Shaderx Series)*, Engel, W. (Editor), Charles River Media, Cambridge, MA, November 2004

[DEMERS05], DEMERS, J. 2005. Depth of Field: A Survey of Techniques, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics,* Fernando, R. (Editor), Addison-Wesley, April 2004.