Real-time Lighting Effects using Deferred Shading

Michal Ferko*
Supervised by: Michal Valient†

Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

Rendering realistic objects at interactive frame rates is a necessary goal for many of today's applications, especially computer games. However, most rendering engines used in these games induce certain limitations regarding moving of objects or the amount of lights used. We present a rendering system that helps overcome these limitations while the system is still able to render complex scenes at 60 FPS. Our system uses Deferred Shading with Shadow Mapping for a more efficient way to synthesize lighting coupled with Screen-Space Ambient Occlusion to fine-tune the final shading. We also provide a way to render transparent objects efficiently without encumbering the CPU.

Keywords: Real-time Rendering, Deferred Shading, High-dynamic range rendering, Tone-mapping, Order-Independent Transparency, Ambient Occlusion, Screen-Space Ambient Occlusion, Stencil Routed A-Buffer

1 Introduction

Our rendering engine is based on concept of Deferred Shading [3], which avoids shading occluded pixels and by postponing the lighting evaluation allows one pixel to be affected by hundreds of lights.

Our system uses HDR rendering coupled with the tone-mapping operator by Reinhard et al. [11] and Bloom, Shadow Mapping [16] to provide hard-edged shadows, Screen Space Ambient Occlusion to simulate indirect lighting and Stencil Routed A-Buffer [8] to render transparent objects. All these techniques allow easy integration into a deferred renderer while providing much more realistic display of scenes.

The main contribution of this paper is a complete rendering pipeline incorporating these well-known techniques. Our aim is to determine in which order these techniques should be applied to avoid incorrect artifacts and how to maintain reasonable quality while allowing real-time display even on older hardware.

*michalferko1@gmail.com
†michal.valient@guerilla-games.com

We are targeting OpenGL 3 capable hardware, because we require the framebuffer object features as well as multiple render targets.

2 Related Work

There are many implementations of Deferred Shading and this concept has been widely used in modern games [15] [12] [5], coupled with techniques used in our paper as well as certain other.

Deferred Shading does not directly allow rendering of transparent objects and therefore, we need to use a different method to render transparent objects. There are several approaches to hardware-accelerated rendering of transparent objects without the need to sort geometry. This group of algorithms is referred to as Order-Independent Transparency.

An older approach is Depth Peeling [7] [4], which requires N scene rendering passes to capture N layers of transparent geometry. Dual Depth Peeling [1] improves the algorithm by capturing two layers of geometry in one pass. However, the objects still need to be rendered multiple times and the performance is still unacceptable for large scenes. Once the layers are captured, a final fullscreen pass blends them together.

A newer approach, Stencil Routed A-Buffer [10], allows capturing of up to 32 layers during one rendering pass thanks to multisample render targets on OpenGL 3 hardware. An additional pass for sorting the values is used. This approach is part of our system.

With OpenGL 4 hardware, it is possible to actually have per-pixel linked lists [13] and thus generate an arbitrary number of layers and sort these samples afterwards in a fullscreen pass. This approach is very similar to Stencil Routed A-Buffer except for the way the samples are stored. We did not use this approach due to the lack of OpenGL 3 support.

To further improve the visual quality of rendered images, we include standard Shadow Mapping [16] for shadow-casting lights and real-time Ambient Occlusion. There has been much research done regarding real-time Ambient Occlusion. In [2], the authors convert polygonal meshes into disks, for which the occlusion computation is simplified and allows dynamic rendering. In [6], the au-

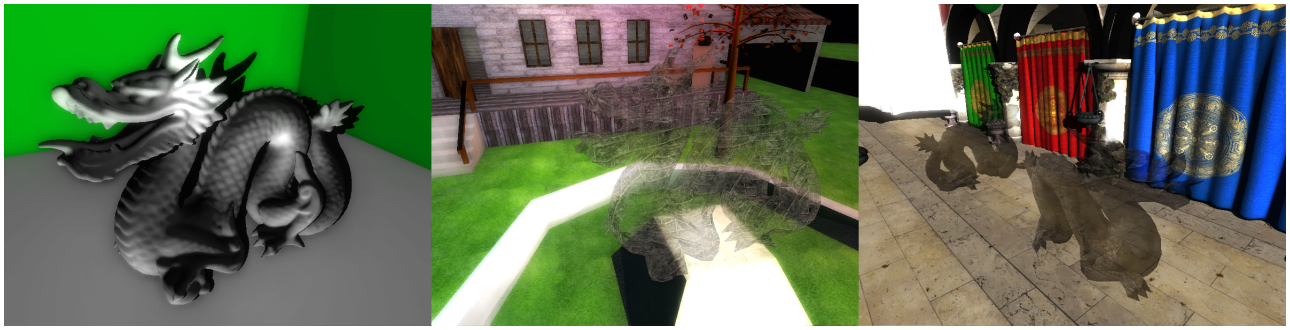

Figure 1: Images of test scenes rendered with our system at interactive frame rates. Dragon scene (Left), House scene (Middle) and Sponza scene (Right)

thors propose a method to generate fields around objects in pre-processing. As long as the objects are not deformed, the fields do not need to be recomputed. It thus allows real-time estimation of occlusion. A similar technique [8] performs a slightly different field generation in the geometry shader and allows for fully-dynamic ambient occlusion even on deformable meshes.

Our work is based on Screen-Space Ambient Occlusion [9] which uses the scene depth information captured during the first stage of deferred shading to approximate scene geometry and thus compute occlusion. The choice was made mainly due to the fact that the previous methods are scene-dependent and perform slower than SSAO when the scene contains hundreds of thousands of triangles. SSAO's performance depends only on the number of samples taken and the screen resolution, being totally independent from the actual scene.

3 Deferred Shading

Deferred Shading [3] is an alternative to Forward Shading, the traditional rendering where a fragment shader accesses all light information at once and outputs the final light contribution directly into the window's framebuffer.

The main idea of Deferred Shading is separation of the lighting calculations from the scene rendering pass (or the geometry pass). During this pass, material and object properties (usually albedo, depth, normal and specular power) are stored into a geometry buffer (G-Buffer).

When compared to forward rendering or multi-pass rendering, the scene is rendered only once and only the fragments that are visible are shaded. No shading needs to be evaluated for objects that are not affected by a certain light (the object is outside of the light volume - part of the scene that is affected by the light).

During the consecutive lighting pass, the light volumes are rendered (cones for spot lights and spheres for point lights) and during the fragment shader execution, the G-Buffer data is read and used to synthesize lighting. The light shapes are rendered with additive blending thanks to the additive nature of light. The results can be displayed on the screen, but usually more post-processing steps are executed after this pass and the results should instead be rendered into a texture - the lighting buffer (L-Buffer).

When rendering light shapes, we use front-face culling to avoid problems when the camera is inside a light volume. Furthermore, for every pixel getting rendered, it is needed to reconstruct the eye-space position corresponding to the current pixel position and the depth stored in the G-Buffer. For this position, we calculate whether it actually is inside the light volume, since we might be rendering nearby light volumes while the G-Buffer contains information about distant objects unaffected by the light.

Deferred Shading mainly outperforms Forward Shading when there are many lights that do not cover large portions of the screen when being rendered. Many directional lights (which affect the entire screen) pose a problem for Deferred Shading, because we do extra work when compared to Forward Shading. Therefore, some implementations evaluate directional lights using Forward Shading [5] and all other lights using Deferred Shading.

In a typical outdoor scene there is usually one directional light representing the sun and in indoor scenes, directional lights are avoided altogether. Our system calculates directional lights with forward shading during the G-Buffer generation phase. Due to this fact, only a fixed amount of directional lights can be forward shaded, for more lights the system would have to switch to deferred shading for the additional lights.

3.1 HDR and Tone-mapping

Having evaluated lighting in a texture adds direct support for HDR rendering. Performing tone-mapping during the L-Buffer generation phase is not possible, since the results get additively blended and we cannot guarantee that a pixel will not be affected by a large number of lights, resulting in a sum of many tone-mapped values which result in a luminance value higher than 1.

Our system is open to many tone-mapping operators. Currently, we are using the global part of the operator by Reinhard et al. [11]. As an input into the tone-mapping stage, we have the L-Buffer which contains 16-bit floating

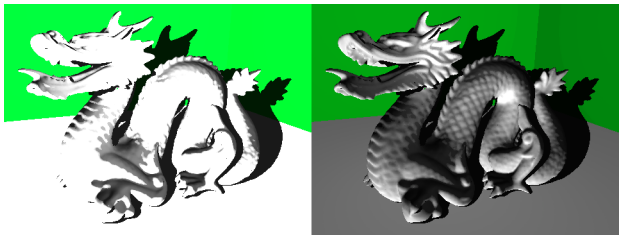

Figure 2: Real-time Reinhard's tonemapping. HDR image before applying tonemapping (Left) and after applying tonemapping (Right).

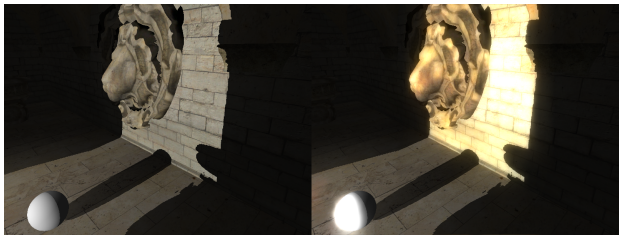

Figure 3: A scene with Bloom disabled (Left) and enabled (Right). Notice the light leaking into the shadow.

point RGB values.

Reinhard's operator analyzes the whole input image and tries to estimate whether it is light or dark. Based on the average luminance of the image L, the luminances are tonemapped into a valid range.

We perform a traditional GPU accelerated calculation of the average luminance of the L-Buffer by downscaling the image up to a 1x1 texture (using mipmap generation) and the value of this one pixel is the average image luminance.

3.2 Bloom

When looking at an object that is occluding a part of a bright light source, the light rays appear to "bend" at the object's edge and a fringe is visible on the occluding object. This effect is called *Bloom*, and it is usually coupled with HDR lighting since only very bright light sources produce such an effect. A scene with and without Bloom is shown in Figure 3.

Our implementation is tightly coupled with the tonemapping operator we use. After calculating the average luminance L of the image, we subtract L from the intensity of pixels in the L-Buffer (we ignore low-luminance pixels) and store these as a tresholded image. Pixels with luminance above average are marked in this image and we perform a blur based on how much brighter the pixels are when compared to average luminance.

Instead of performing a gaussian blur of different kernel size in each pixel, we chose to generate mipmaps for the tresholded image and sum values from multiple mipmaps, while alpha blending into the final image based on resulting pixel luminance.

3.3 Shadow Mapping

Incorporating Shadow Mapping into a Deferred Renderer is straightforward and allows for hard-edged shadow without much effort.

Shadow Mapping [16] is a fully GPU accelerated method for rendering real-time shadows. The entire scene is rendered one additional time (from the light's point of view in the light's direction) for every shadow-casting light, storing the depth values generated during this rendering.

Afterwards, when evaluating the lighting equation, the world position of the current surface point is projected (using the same projection as was used during the shadow map generation phase and remapping from $[-1,1]^3$ range into $[0,1]^3$) which gives us the (x,y) coordinates in the shadow map and a depth value z. The fragment shader reads the depth value d at position (x,y) in the shadow map and compares it to z. If $z = d$, the point with depth z is the closest point to the light source and therefore is not shadowed. If $z > d$, this point is behind a point with depth d and is therefore not directly lit by the light.

Due to floating-point precision errors, a small offset needs to be added to all values stored in the shadow map, otherwise non-realistic self-shadowing artifacts occur.

Integrating Shadow Mapping into a Deferred Renderer is simple. The scene is rendered one additional time for every shadow-casting light and thanks to the fact that we only access one light at a time in the fragment shader, we can reuse one texture for multiple lights. Our system does this by flip-flopping between shadow map generation and L-Buffer generation. We first generate the shadow map for the first light, then render the light's shape into the L-Buffer while accessing the map. We then clear the shadow map, render from the second light's point of view (into the same shadow map), and render the second light's shape into the L-Buffer. For lots of shadow-casting lights, this is a necessary approach due to the fact that we already took up a lot of memory with the G-Buffer (and the L-Buffer).

4 Transparent Objects

The Deferred Shading approach does not support rendering of transparent objects. The G-Buffer contains information only about the nearest pixels, but we require multiple points per pixel when rendering transparent objects to correctly compute the final pixel color.

Due to the alpha blending equation:

$$color_{final} = (1 - \alpha_{src})color_{src} + \alpha_{src}color_{dst}, \quad (1)$$

which is used for correct rendering of transparent objects, the resulting color needs to be evaluated back-to-front from the camera's point of view. Therefore, a typical implementation sorts all triangles in the scene and then renders these back-to-front. Problems are pairs of intersecting

triangles which introduce the need to split at least one of those triangles into two parts.

For static transparent objects, constructing a BSP tree as a pre-processing step for all transparent triangles in the scene is a common approach. During scene rendering, the BSP tree is traversed back-to-front based on camera location. However, when we consider dynamic objects, the BSP tree needs to be maintained every frame (in worst-case, the whole tree needs to be rebuilt), which is usually a CPU intensive approach, especially for thousands of triangles.

4.1 Stencil Routed A-Buffer

Our goal was to have fully dynamic scenes, therefore we chose Stencil Routed A-Buffer [10] for rendering transparent objects. Thanks to newer hardware supporting OpenGL 3 and higher, it is possible to render to a multisample framebuffer object (FBO) and use the stencil buffer to fill each sample of a pixel with different values at different depths. This feature is called *Stencil Routing*.

At first, a multisample framebuffer with a depth, stencil and color buffer is created - this will be our Alpha buffer (A-Buffer). Then, every frame, the stencil values are initialized to $n+1$ for the n-th sample of a pixel by rendering a full-screen quad once for every sample while allowing writing only into the current sample.

During the rendering of transparent objects (with depth write disabled), we set the stencil operation to decrease whenever a fragment is being rendered and we set the stencil function to equal with the reference value 2. When the first fragment of a pixel is being rendered, the first sample (which has a stencil value of 2) is filled with color and depth of the fragment and all stencil values are decreased by one. Now the second sample has a stencil value of 2 and the next fragment being rendered for this pixel gets stored in the second sample. This behavior is shown in Figure 5.

Using this approach, we can compute n layers of transparent objects in one rendering pass, where n is the number of samples per pixel in our multisample FBO. Latest hardware allows up to 32 samples per pixel, but older graphic cards support 8 samples with no problems.

Finally, we need to display the transparent objects on the screen. We render a fullscreen quad and in the fragment shader, we access all the samples one by one and sort them based on their depth value. Finally, the sorted samples are blended in the fragment shader using standard alpha blending and the result is displayed on the screen. A result is shown in Figure 4.

Despite the improvement on speed of rendering, there is still a problem with shading transparent objects. When blending the fragments together, we need to shade them accordingly, and this is a forward shading step, so it inherits all the limitations of forward shading. Therefore, only a small amount of lights can be selected for shading the transparent surfaces.

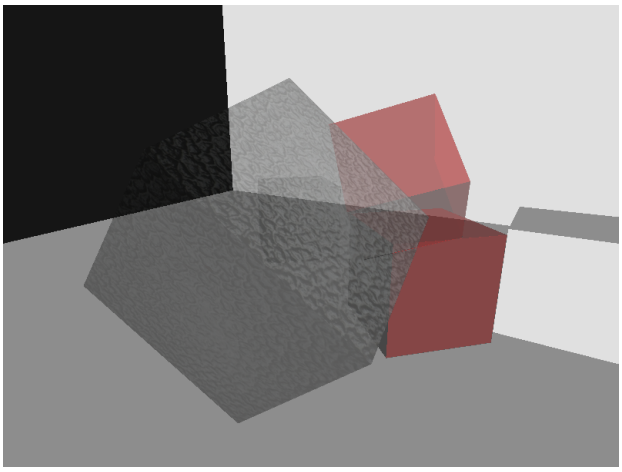

Figure 4: Stencil Routed A-Buffer - Result

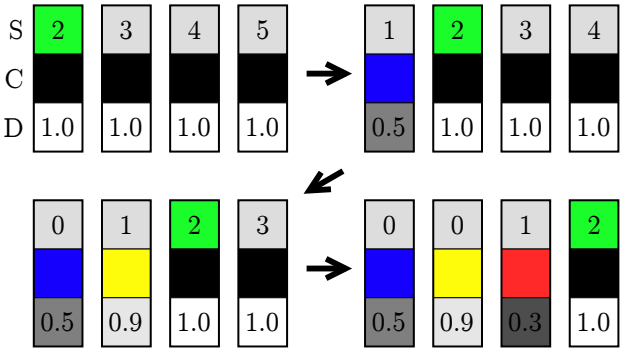

Figure 5: Stencil Routing - The 4 samples for a pixel after initialization step ready to write to the sample with stencil value $S = 2$ (top-left). When the first fragment is stored all stencil values are decreased by one, color C and depth D of the first fragment are written into the first sample (top-right) and after next two steps (bottom-left, bottom-right) we have three valid samples that can be blended together.

Problems can still occur when there are more samples per pixel than the multisample buffer can hold. Samples get lost and there is the question of what to do when an overflow occurs. One simple improvement is to use bounding volumes which are probably used during view frustum culling to estimate distance from camera and render sorted (not completely like with a BSP tree) geometry, which causes the closest surfaces to be stored in our A-Buffer before it overflows. Afterwards, overflown pixels will probably be more distant and their final contribution might be so small that the artefacts will not be visible.

5 Ambient Occlusion

Ambient occlusion is defined as the amount of ambient light reaching a point in the scene. It is calculated by integrating the visibility function over a hemisphere centered at the target point and oriented according to the surface

normal. Static ambient occlusion can be precomputed for static objects (and dynamic lights, since the occlusion does not depend on light sources), however it is a costly process and it does not produce correct results when there are moving objects in the scene.

5.1 Screen-Space Ambient Occlusion

Dynamic pre-computation of ambient occlusion is impossible especially when we don't know how objects in the scene will move. Therefore, we need a dynamic solution that estimates occlusion every frame.

In Mittring's work [9], the author proposed a method called Screen-Space Ambient Occlusion (SSAO) which uses the depth buffer (as rendered during the G-Buffer phase) to approximate scene geometry and estimate occlusion based on this inaccurate approximation.

The output of the algorithm is a one-channel 8-bit *accessibility texture* covering the entire screen. The value in each pixel in range $[0,1]$ is then used as a multiplication factor for ambient light of the respective pixel.

The SSAO generation pass occurs after the G-Buffer is generated and before we synthesize lighting, since we want to use the occlusion values during the lighting phase. We render a fullscreen quadrilateral and access the G-Buffer depth and the G-Buffer normal. For every pixel p with (x,y) coordinates and d depth, its 3D eye-space position P is reconstructed using the depth value and position in the depth map. In a small sphere around this 3D point, a number of points $Q_0,...,Q_c$ are generated by adding random vectors of length less than 1 multiplied by the sphere's radius r to the point's position.

Afterwards, every point Q_i is projected back into clip space which gives us (x_i,y_i) coordinates to the depth map and the actual depth d_i of point Q_i. If the value stored at position (x_i,y_i) in the depth map is smaller than d_i, there is an object covering point Q_i and it is a potential occluder for point P.

Our approach utilizes the G-Buffer normal as well, which increases the number of correct samples - samples that are below the surface are false occluders. This is not included in the Crytek implementation and it avoids self-occlusion which generates occlusion values of 0.5 on flat surfaces. In Figure 6, we have the scene's depth buffer as seen from the camera C. Point Q does not lie in the half-space assigned by point P and the normal at P. Therefore, any object containing Q should not be considered as an occluder. The object in front of Q should be considered and we rely on the point distribution that at least one of the sampling points will be inside the sphere and the sphere will contribute to occlusion. Even if this is not the case for one point, for neighbouring pixels it probably will be and after blurring the occlusion value gets propagated from those as well.

To avoid generating points behind the surface, we test the angle between the surface normal and the offset vector

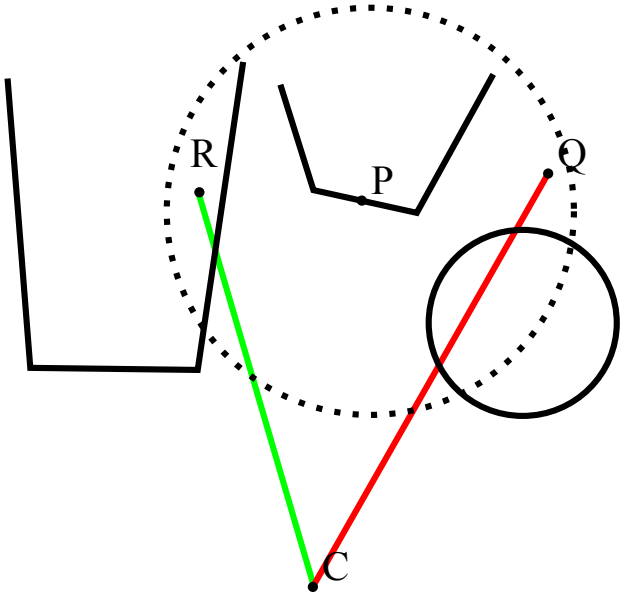

Figure 6: The SSAO algorithm - When calculating occlusion for the surface point P, we generate points such as Q and R in the dotted sphere with radius r. Both Q and R have depth larger than the value stored in the depth buffer, therefore both points are considered as occluders.

$\vec{v}$ added to P. If larger than $\frac{\pi}{2}$ we simply take $-\vec{v}$ as the offset vector.

For every potential occluder, the occlusion factor is computed as a function of the occluder's distance d:

$$O(d) = \frac{1}{1+d^2}. \tag{2}$$

A necessary step during the SSAO computation is the generation of random vectors that are added to P. If for every pixel we use the same set of vectors, certain repeating patterns occur. A more acceptable solution is that the vectors differ for neighboring pixel. We use a RGB texture containing random values that are interpreted as a vector and up to 32 uniformly distributed vector (stored as uniform variables during shader execution) in a unit sphere. In the shader, all the uniformly distributed vectors are reflected by using the vector value read from our random vector texture. This ensures different rotation of vectors on the sphere for neighboring pixels.

The difference can be seen in Figure 7. The randomized pattern gets blurred into a smoother result since it avoids large portions of uniformly occluded areas, which would not disappear after blurring.

The SSAO fragment shader performs a lot of operations. However, the results are quite noisy even when using a large number of samples and they still require a blurring step. Therefore, using a $\frac{w}{2} \times \frac{h}{2}$ (where $w \times h$ is the current resolution) sized SSAO texture is acceptable because it will be blurred anyway. Some implementations even use $\frac{w}{4} \times \frac{h}{4}$ SSAO textures.

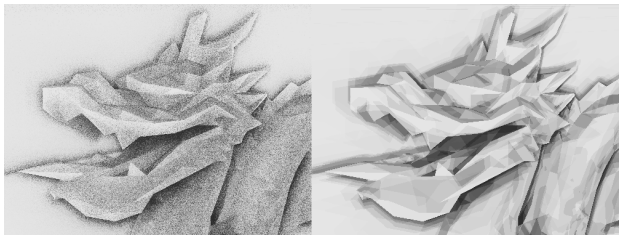

Figure 7: The difference when using randomized vector in SSAO (Left) and same for each pixel (Right)

5.2 Cross-Bilateral Filtering

Actual occlusion is mostly smooth and without any kind of noise. Therefore, we need to reduce noise somehow. Simply blurring the SSAO texture with a gaussian blur is not enough, because occlusion will "leak" through edges in the screen which results in an unnatural behavior such as the detaching of the occlusion in Figure 8.

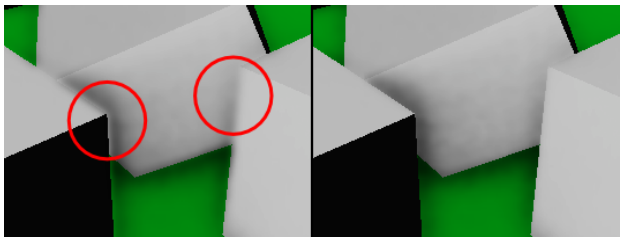

Figure 8: Artifacts occuring when using gaussian blur (Left) and elimination of those artifacts when using Cross-Bilateral Filtering (Right)

To avoid blurring over edges, we use a modified gaussian blur. What we want is an edge-aware blur that does not blur edges in the image. A great help is the depth buffer we have stored. The difference of depth values between two neighbouring pixels describes quite well whether there is an edge between these two pixels or not. If there is a tiny difference, the neighbouring pixels were very close to each other before projection.

We use a version of cross-bilateral filtering [14] coupled with a separable gaussian blur. We perform a horizontal and vertical blurring pass on the SSAO texture while multiplying the gaussian coefficients with a simple function of depth difference between the center sample d_{center} and the current sample $d_{current}$. We use the following function:

$$w(current) = \frac{1}{\delta + |d_{center} - d_{current}|}, \qquad (3)$$

$\delta > 0$ is a small constant to avoid division by zero.

After multiplying all samples with their corresponding weights, the sum of all weights is computed so we can normalize the result.

The results of SSAO after performing a simple gaussian blur and a cross-bilateral blur are shown in Figure 9.

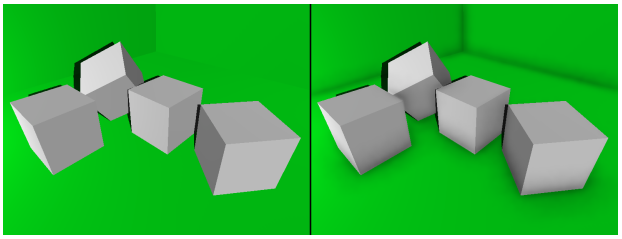

Figure 10: Comparison of a final scene without SSAO (Left) and with SSAO (Right)

In Figure 10, you can see the visual improvement provided by SSAO with a 15x15 cross-bilateral filter.

6 Complete rendering pipeline

Our implementation was written in C++ and OpenGL and runs on hardware supporting OpenGL 3.3. When putting together all the rendering parts described in this paper, the steps of our pipeline are as follows:

1. Render non-transparent objects while generating the G-Buffer.
2. Using the G-Buffer depth and normal textures, generate the SSAO accessibility texture.
3. Render transparent objects using Stencil Routed A-Buffer into a multisample buffer and perform forward shading on all pixels with the selected lights.
4. Render light volumes while generating the L-Buffer. Access SSAO texture for ambient term.
5. Blend sorted samples from the multisample buffer into the L-Buffer.
6. Compute log-average luminance and L_{white} from the L-Buffer.
7. Prepare unfiltered bloom texture by subtracting values based on average luminance.
8. Compose final bloom image by averaging multiple mipmaps.
9. Tonemap L-Buffer and display on screen.
10. Display Bloom on top of tonemapped data.

7 Performance and Results

We tested our rendering engine on several scenes, the testing machine had an Intel Core i5 750 CPU, a NVIDIA GeForce 9800 GT graphic card and 4GB of RAM. Our application is single-threaded at the moment, it was utilizing one core of the CPU.

We used three different quality settings of our engine during the tests:

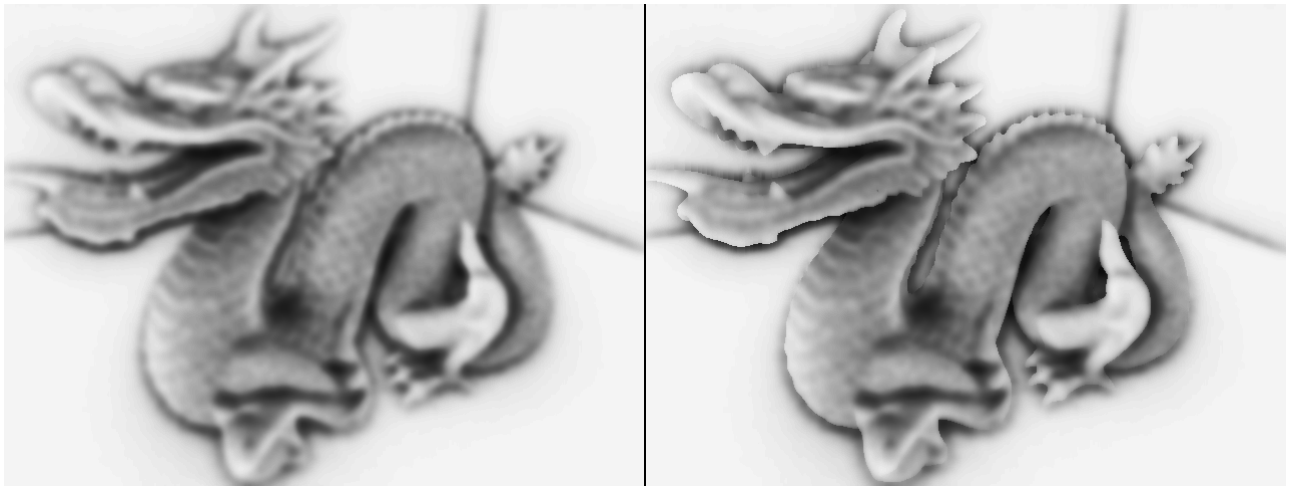

Figure 9: Comparison of simple blurring (Left) and edge-preserving blurring (Right) of the accessibility texture. Both images have been darkened for the effects to be visible. Most notable differences are around the dragon's head

- **Low** - 800x600 resolution, using $\frac{1}{16}$ sized SSAO texture with 16 samples per pixel and a 21x21 pixel wide bilateral blur. Stencil Routed A-Buffer had 4 samples per pixel. Using one 512x512 shadow map for sunlight.
- **Medium** - 1280x720 resolution, using $\frac{1}{4}$ sized SSAO texture with 16 samples per pixel and a 61x61 pixel wide separable bilateral blur. Stencil Routed A-Buffer had 8 samples per pixel. Using one 1024x1024 shadow map for sunlight.
- **High** - 1920x1080 resolution, using full-sized SSAO texture with 16 samples per pixel and a 61x61 pixel wide separable bilateral blur. Stencil Routed A-Buffer had 8 samples per pixel. Using one 2048x2048 shadow map for sunlight.

Furthermore, we used multiple presets to determine which operations take how much performance.

- **SSAO + SR** - SSAO and transparent objects enabled
- **SSAO** - SSAO enabled and transparent objects disabled
- **SR** - SSAO disabled and transparent objects enabled
- **OFF** - SSAO and transparent objects disabled

The results are shown in Table 1, listing average FPS values. SSAO is the main time-consuming operation, especially on High settings when using a full-sized buffer. However, on lower settings, the effects of SSAO were still visually appealing and only an experienced user would notice some artifacts due to the low resolution of the buffer. We do not recommend doing SSAO in full-size, $\frac{1}{4}$ sized buffer is a good performance and quality trade-off.

Note that the performance of SSAO depends mainly on screen resolution and samples per pixel. Since it is a screen-space algorithm, it does not in any way depend on scene complexity. This is one of the advantages of SSAO, that it provides stable performance and no unexpected FPS spikes.

Stencil Routed A-Buffer, on the other hand, depends on how many transparent objects and layers are visible at the moment. The more pixels, the more we need to sort and the higher FPS spikes. In the Dragon scene, there were no transparent objects, therefore the measurements are omitted.

All testing scenes had 6 animated point lights and one static directional light with Shadow Mapping as a good base for Deferred Shading.

8 Conclusion

We have presented a fully-dynamic real-time rendering system that overcomes any kind of pre-processing steps and allows dynamic objects and lights. The system runs at interactive frame-rates on newer hardware but it is compatible with OpenGL 3 hardware and it can be altered (in terms of quality) to run at interactive frame rates on older hardware as well. We have presented how the different techniques fit together and provide visually appealing quality.

Our system still has limitations to overcome, especially allowing an arbitrary number of lights to affect transparent objects without a performance hit. Other Real-time Ambient Occlusion techniques as well as Per-pixel Linked Lists for Order Independent Transparency should also be integrated into the system to evaluate the quality/speed trade-off and provide other solutions for latest hardware.

Scene	Triangles	Preset	Low	Medium	High
Sponza	287 353	SSAO + SR	18.7	10.9	3.0
		SSAO	20.9	13.8	3.6
		SR	19.7	15.8	10.7
		OFF	24.7	21.3	14.6
Dragon	201 075	SSAO	38.1	19.1	3.8
		OFF	49.6	36.2	21.2
House	11 563	SSAO + SR	46.7	18.3	3.9
		SSAO	60.1	26.0	4.6
		SR	67.5	32.7	17.8
		OFF	95.0	60.0	36.3

Table 1: Overall performance on three different scenes. Showing average FPS values for different settings of our engine

9 Acknowledgements

We would like to thank Marko Dabrovic for providing the Sponza Atrium model and The Stanford 3D Scanning Repository for providing the Dragon model. We also thank Martin Madaras and Andrej Ferko for providing the hardware for testing.

References

[1] L. Bavoil and K. Myers. Order independent transparency with dual depth peeling. Technical report, NVIDIA Developer SDK 10, February 2008.

[2] Michael Bunnell. Dynamic ambient occlusion and indirect lighting. In Matt Pharr, editor, *GPUGems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 223–233. Addison-Wesley, 2005.

[3] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: a VLSI system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22:21–30, June 1988.

[4] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, May 2001. Available at http://www.nvidia.com/.

[5] Dominic Filion and Rob McNaughton. Effects & techniques. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 133–164, New York, NY, USA, 2008. ACM.

[6] Janne Kontkanen and Samuli Laine. Ambient occlusion fields. In *Proceedings of ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pages 41–48. ACM Press, 2005.

[7] Abraham Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9:43–55, July 1989.

[8] M. McGuire. Ambient occlusion volumes. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 47–56, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[9] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM.

[10] Kevin Myers and Louis Bavoil. Stencil routed a-buffer. In *ACM SIGGRAPH 2007 sketches*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[11] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 267–276, New York, NY, USA, 2002. ACM.

[12] O. Shishkovtsov. Deferred shading in S.T.A.L.K.E.R. In Matt Pharr and Fernando Radima, editors, *GPU Gems 2*, chapter 9, pages 143–166. Addison-Wesley Professional, 2005.

[13] Nicolas Thibieroz and Holger Gruen. OIT and indirect illumination using DX11 linked lists. In *GDC San Francisco 2010*.

[14] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society.

[15] Michal Valient. Deferred rendering in killzone 2. Online, accessed Feb. 20th, 2012, 2007. Develop Conference, http://www.guerrillagames.com/publications/dr_kz2_rsx_dev07.pdf.

[16] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12:270–274, August 1978.