Stochastic Depth of Field using Hardware Accelerated Rasterization

Authors	*Examiner*	*Advisors*
Robert Toth	Tomas Akenine-Möller	Jacob Munkberg
Erik Linder		Jon Hasselgren

Lund University

Abstract

Depth of field is a desired, but computationally expensive effect in computer graphics. A number of algorithms exist, each having distinct drawbacks or limitations. Accumulation buffering is readily hardware accelerated but requires much computation time in order to eliminate unnatural artifacts. Post-processing techniques are popular because of their performance but are inaccurate. Stochastic ray tracing gives excellent quality but is computationally intense and not easily accelerated using contemporary graphics hardware.

In this thesis, we implement stochastic rasterization on contemporary GPUs, and it is shown to be as efficient as accumulation buffering with comparable image quality with less obvious artifacts. We show that it is already viable to use stochastic rasterization in real-time applications, although several graphics hardware optimizations are unavailable using this technique. Some of these optimizations can be extended to support stochastic rasterization with only slight hardware modifications, making the proposed method likely to outperform accumulation buffering for depth of field effects.

Contents

1 Introduction

1.1 Stochastic rasterization

In this thesis, we implement stochastic rasterization [Cook et al. 1984] for depth of field effects on contemporary graphics processing units (GPUs), and it is shown to be as efficient as accumulation buffering with comparable image quality with less obvious artifacts. We show that it is already viable to use stochastic rasterization in real-time applications. In combination with a good mesh reduction system, stochastic rasterization outperforms accumulation buffering, becoming the fastest correct real-time depth of field method to date.

1.2 Depth of field

In photography, certain objects are sharp, while others are blurry. This can be caused by objects moving relative to the camera during exposure, or because of objects being closer or more distant than the focal distance. The former effects are called *motion blur* and will not be discussed here. The latter is the effect called *depth of field*.

Depth of field occurs as soon as a three-dimensional object is being projected onto a flat surface through a lens of non-zero radius. This means that all real-world optical systems — most notably cameras, and even human eyes — experience some degree of depth of field.

With proper care, depth of field can be used to direct the attention of a viewer to specific areas of an image by making surrounding details less prominent and also to enhance the sense of depth in the scene. This is demonstrated in Figure 1, where the same scene is shown with two different focal depths. In the left image, the attention is directed towards the buildings in the background. In the right image the attention is naturally focused on the statue in the foreground as it is the only thing depicted sharply.

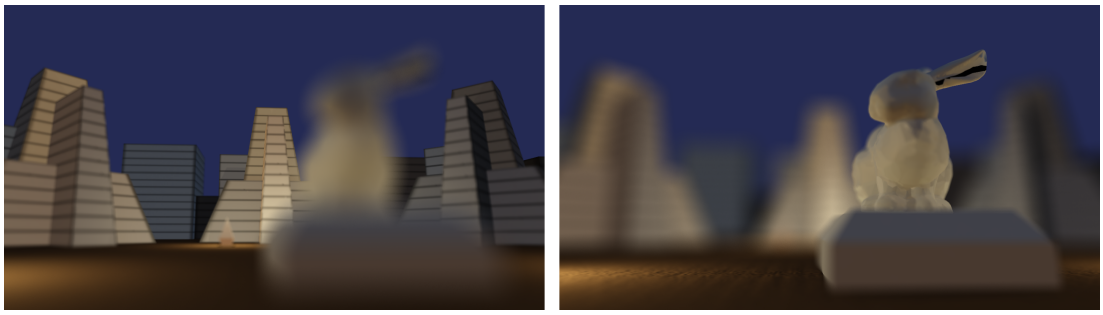

Figure 1: *Directing attention using depth of field.*

Another interesting area of application is matching a virtual camera to a real world camera in order to merge a film clip with a computer generated scene. A human would easily recognize a sharp CG object in the merged footage if the real film clip contains perceivable depth of field effects.

1.3 Mathematical model of a camera

Computer graphics in general does not take depth of field in to account and simulates the virtual camera with an idealized pinhole camera. This means that the lens has no size and light can only travel through one single path through the scene towards the camera film.

When projecting a scene through a lens of non-zero radius, the focal length f of the lens and the distance a between the lens and the image plane determines the distance z_0 to the plane in focus by the lens equation:

$$\frac{1}{f} = \frac{1}{a} + \frac{1}{z_0}$$

Any single point at distance z_0 will be projected to a single point on the opposite side of the lens at distance a. Light from a point at distance z_0 from the lens will thus converge to a single point on the image plane. If another point, at distance z, is projected through the same lens, it will be projected to another point of distance b from the lens. Light arriving from a point of distance z to the entire lens area will converge towards a point of distance b, and thus cover an entire area of the image plane. This is illustrated in Figure 2. The result is a smudged image, where the extents of the smudge is called the *circle of confusion (CoC)*.

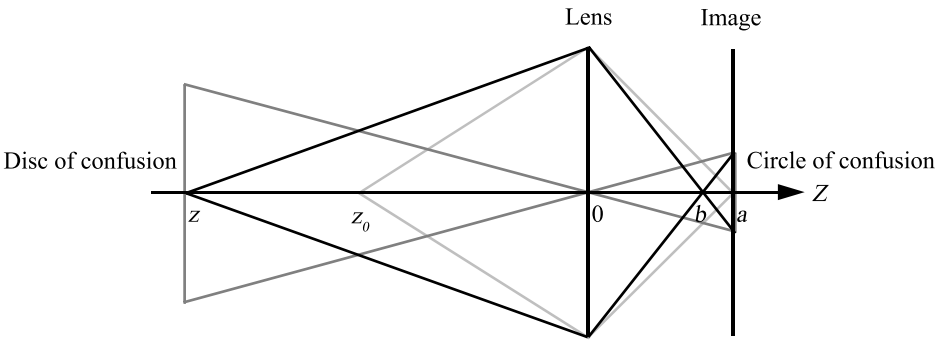

Figure 2: *Confusion projections through a lens. Light from point z_0, shown in light grey, is sharply projected onto the image plane. Light from z, shown in black, covers a circle of confusion on the image plane. The same result is obtained by projecting an equivalent disc of confusion through a pinhole camera, shown in dark grey.*

Given the aperture size A, the circle of confusion radius in the image plane can be calculated as [Demers 2004]

$$CoC = Af\frac{|z - z_0|}{z(z_0 - f)}.$$

Since we will be working mostly in 3D space in later chapters, we introduce the following concept: The *disc of confusion (DoC)* is a circular disc in 3D space, aligned along the camera viewing direction that a point must be evenly smudged across to achieve the same visual result, using a pinhole camera, that would be obtained by the non-smudged point with a real-lens camera. This distinction is illustrated in Figure 2. We reverse the projection by re-multiplying the coordinates by the object distance z and rewrite the circle of confusion equation as

$$DoC = Af\frac{|z - z_0|}{z_0 - f}$$

with the same constants as above. Unlike the circle of confusion radius, the disc of confusion radius varies linearly across a triangle face, since the depth is linear along the triangle surface. All the parameters can be accurately calculated to match a specific camera. They can then be grouped together into two variables k and z_0 to simplify calculations:

$$k = \frac{Af}{z_0 - f},$$
$$DoC = k(z - z_0).$$

Correctly applying the mathematical model described above using any method to a triangle-based 3D-model will yield several noteworthy characteristics: A single solid triangle will be internally blurred, and also get a semi-transparent region according to the circle of confusion. Several connected triangles will smoothly melt together at the seam, without any transparency. Any triangle not directly facing the Z-axis will get a varying circle of confusion along the face. Any object behind an out-of-focus occluding object will be partially visible through the semi-transparent region of the occluding geometry.

2 Current techniques

2.1 Ray-tracing

Ray-tracing is a technique for rendering scenes by repeatedly casting rays from a virtual camera through each pixel of the rendered image. For each ray, the intersection point with the scene geometry is determined and the pixel color is calculated using this point. Depth of field is accomplished by distributing several rays across the lens for each pixel and directing them as to converge according to the lens equations [Cook et al. 1984]. Since the true light transport is simulated, the image acquired this way is unbiased and will converge towards the correct image with an increasing amount of rays.

Ray-tracing is almost exclusively used for high quality offline renderings as it is a very time consuming process that is not accelerated by current graphics hardware.

2.2 Accumulation buffering

Accumulation buffering techniques (ABT) is based on the pinhole camera model. Instead of rendering a single image of the scene, a number of different locations on the lens are used as camera positions to achieve depth of field. The rendered image from each of these positions is weighted and blended together to create the final image [Haeberli and Akeley 1990]. The accumulated image is unbiased and converges towards the correct image with an increasing number of render passes.

Artifacts appear with this technique when too few samples are used; a distinct outline of every sample image is visible in the accumulated result, shown in Figure 3. This is called *ghosting*.

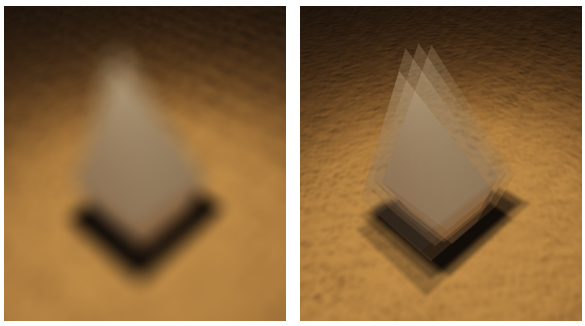

Figure 3: *Left: reference image, 256 sample stochastic rasterization. Right: accumulation buffering using 4 samples, showing severe ghosting.*

Ghosting is most visible in the blurriest regions, meaning that the largest circle of confusion of the entire image decides the number of passes needed for a certain overall image quality. The number of passes needed is proportional to the circle of confusion area in pixels.

Contemporary hardware is well utilized by this technique, since each pass is rendered using the standard pinhole camera. A drawback is that the geometry of the entire scene is sent to the GPU once per lens sample [Wexler et al. 2005]. For real-time applications it is important to minimize the amount of draw-calls, otherwise the performance may become CPU-bound [NVIDIA 2006]. Accumulation buffering intensifies this problem since the amount of draw-calls scales linearly with the number of passes.

2.3 Post-processing

Depth of field can be approximated by doing a pinhole camera rendering and subsequently blurring the rendered image with a variable blur kernel according to the depth of each pixel. The images produced this way are severely biased due to the fact that they only consider light arriving at a single point of the camera lens.

Many artifacts appear with post-processing depth of field techniques [Demers 2004]. One of these is bleeding of blurry background objects into a sharp foreground. Another severe but typical artifact is that blurry foreground objects fail to blend over sharp backgrounds.

Post-processing techniques are still often used in real time applications due to their ability to produce relatively appealing visual results at a low constant cost.

2.4 Stochastic rasterization

Stochastic rasterization (SR) was introduced by Cook et al. [Cook et al. 1984], and depth of field was described as one of its potential uses. It was later implemented on a GPU [Wexler et al. 2005], but was found inferior to accumulation buffering with their naïve implementation. Time-continuous triangles [Akenine-Möller et al. 2007] were introduced in order to make stochastic rasterization for single-degree-of-freedom systems, such as motion blur, efficient on graphics hardware.

Akenine-Möller et al. proposed a hybrid accumulation buffered stochastic rasterization technique in order to achieve real-time depth of field. In contrast, we present a single-pass stochastic rasterization technique.

3 Workflow of our technique

High quality is best achieved by means of stochastic rendering. However, typical graphics hardware is highly optimized for pinhole-camera rasterization. Our method combines the strengths of both into a single technique. To conform to contemporary GPUs, a standard pinhole camera is the basis the for rasterization. The typical real-time approach to achieve true depth of field, as described in chapter 2.2, perturbs the pinhole camera to numerous locations on the camera aperture. Our approach instead perturbs the rendered geometry.

In this chapter, the layout and functionality of a typical contemporary graphics hardware pipeline is first described, followed by an overview of how our method is applied throughout the pipeline.

3.1 Pipeline of contemporary graphics hardware

When data is passed from the application to the graphics hardware, it is sent as a stream of vertices. These vertices are first processed individually in the *vertex shader*. The vertex shader processes a single vertex at a time and always output one vertex per input vertex.

When all the vertices belonging to a single triangle are processed by the vertex shader, they can optionally be passed to a *geometry shader*. The geometry shader can perform per-triangle calculations as it has access to all vertices comprising a triangle at once. It may also do data amplification, outputting more than one triangle for each input triangle. If no geometry shader is used, the input triangle is directly passed to the next stage of the pipeline.

Each triangle is then rasterized by fixed-function (non-programmable) hardware. Rasterization is the process of determining which pixels, also referred to as *fragments*, are covered by a triangle. The input positions to the rasterizer, and thus the output from the previous shader, are expected in clip space. Each rasterized pixel invokes the *pixel shader*, also called the *fragment shader*. The pixel shader has access to parameters outputted from earlier steps, linearly interpolated across the triangle in homogenous space. The pixel shader must always output a color and may optionally output a depth value. If no depth is written in the pixel shader, it is instead interpolated from the vertex depths in homogenous space.

Incorporating our proposed technique in these stages is accomplished by using each of the three programmable shaders as described in the following subchapters. An illustration of the workflow can be seen in Figure 4.

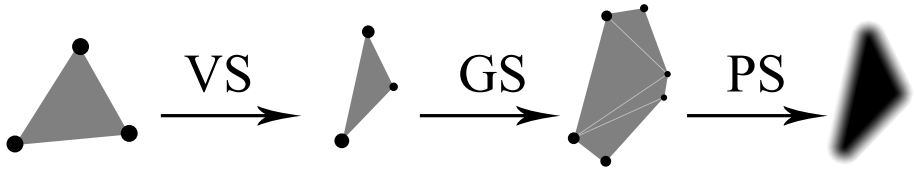

Figure 4: *The triangle-processing workflow. The vertices are first processed individually by the vertex shader (VS). The triplet of vertices forming a triangle are then processed by the geometry shader (GS). Finally each pixel covered by the GS output is processed by the pixel shader (PS).*

3.2 The vertex shader

The main role of the vertex shader in our technique is to perform transformation from object space to view space. The only additional information that can be calculated per vertex is the disc of confusion radius since this varies linearly along the view vector.

Working in view space is convenient since it simplifies later calculations as it is a 3D space with the camera aligned to the Z-axis. All blur thus takes place in the x and y directions, with magnitude depending only on the z components.

3.3 The geometry shader

By calculating our proposed disc of confusion in 3D-space, a bounding volume V can be calculated for each triangle by enclosing the circular disc DoC for each point q on the triangle surface T, also illustrated in Figure 5:

$$V = (p : p \in DoC(q),\ q \in T) \subset \mathbb{R}^3$$

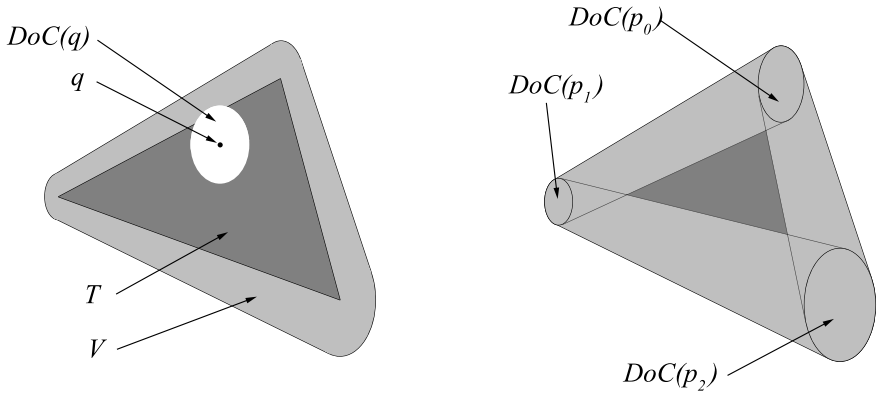

Figure 5: *Left: exact bounding volume (light grey) can be seen as a generalized Minkowski-addition of the original triangle (dark grey) and the variable disc of confusion (white). Right: The dark area is seen as solid, as it will be covered for every valid perturbation of the triangle T. The light area is not covered for all perturbations and is thus semi-transparent.*

Stochastic intersection testing — which will be further explained in chapter 5 — can only hit the triangle within the projected area of this volume. In order to obtain an image of the entire blurred triangle, the projection of the bounding volume needs to be rasterized.

We propose two approaches to calculate triangle confusion bounds. The first approach is to construct a conservative bounding volume in 3D space around the exact bounding volume. The second approach is to calculate a bounding area that covers the projected area of the enclosing volume in image space. Both result in data amplification; for each triangle input, a set of triangles are generated, forming the bounding volume or area. The data amplification is generally higher when constructing three-dimensional volumes.

Each of the output triangles contain all vertex attributes of the contained triangle: the location of the three unperturbed corners along with their associated parameters such as texture coordinates and normals. The data regarding the contained triangle is stored as view space coordinates, while the positions for the bounds are in clip space since the graphics hardware requires this for the fixed-function rasterization stage.

3.4 The pixel shader

The pixel shader is executed for each pixel of the bounding volume generated by the geometry shader. Since each triangle of the bounding volume contain all data regarding the enclosed triangle, stochastic sampling is made possible. The details of this procedure is covered in chapter 5.

Since each pixel is sampled only once, the cast ray will either hit or miss the triangle in the semi-transparent region seen in Figure 5. The result is a jitter between the triangle and the background. Even in the center of the triangle, where a hit often is guaranteed, adjacent fragments will not sample the surface coherently and thus give rise to internal jittering as well.

Noise reduction is crucial to all stochastic methods since they have a high variance by nature. This is solved by super sampling which is explained in chapter 9.4. Mathematical statistics states that four times

as many samples are required to halve the standard deviation. Assuming that the distance of rendered objects is large compared to the lens radius, the incoming light to the camera will be evenly distributed across the lens surface. The stochastic sampling should thus be uniformly distributed[1]. If the aperture is circular, the perturbed position can be calculated as follows:

$$\begin{cases} u_0 \in [0,1) \\ u_1 \in [0,1) \end{cases} \rightarrow \begin{cases} r = \sqrt{u_0} \\ \theta = 2\pi u_1 \end{cases} \rightarrow \begin{cases} x = r\cos\theta \\ y = r\sin\theta \end{cases}$$

The stochastic variables x and y representing the position on the lens are calculated from two uniform distributions that can be stratified to prevent sample clustering, further reducing noise.

[1] Unless time-varying aperture effects are desired, which is discussed in chapter 5.3

4 Bounding volumes

In this chapter, several different methods for creating bounding volumes are discussed. The strong and weak points of each technique are considered.

It is important to minimize the unnecessary rasterized area without performing excessive calculations and using too much bandwidth between the geometry shader and rasterizer. Any area outside the true enclosing volume will always result in discarded pixels and wasted computation time. The bandwidth used is proportional to the number of output vertices from the geometry shader. Different workloads result in bottlenecks at different parts of the graphics pipeline, as will be discussed in chapter 7.1.

This chapter will refer to different bounding volume terms. We define them as follows, also illustrated in Figure 6:

- The *exact bounding volume* is defined as all points belonging to the enclosed triangle for some perturbation.
- The *convex bounding volume* is the smallest convex bound for the exact bounding volume. Another interpretation of the convex bounding volume is the convex bound for the discs of confusion of the enclosed triangle's vertices.

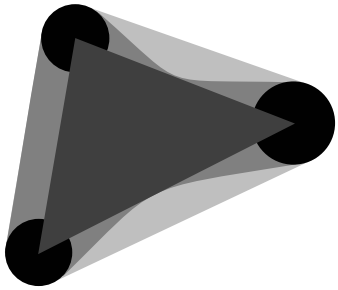

Figure 6: *Bounding volumes. The black circles are the discs of confusion at the three vertices of the dark grey triangle. The medium grey area is the exact bounding volume, and the light grey area is the convex bounding volume. The bounding volumes include the darker areas in the figure.*

The narrow waist shown in Figure 6 appears when a triangle spans across the focal plane. If the triangle is entirely on one side of the focal plane, then the exact and convex bounding volumes will be identical.

All of our bounding volumes can be constructed in a single triangle strip, meaning the number of output vertices is always the number of triangles plus two.

4.1 Overview of bounding volumes

Class	Bounding method	Vertices	Triangles	Computation	Usability
3D	3D Hexagon	12	20/16	heavy	excellent
3D	Wedge	6	8	medium	good
3D	Reduced Wedge	5	4	intense	excellent
2D	2D Hexagon	6	4	medium	mediocre
2D	Parallel edge triangle	3	1	light	poor
2D	Variable radius triangle	3	1	medium	very poor

Table 1: *Summary of bounding volumes*

Table 1 shows an overview of the relevant bounding solutions discussed in the subsequent subsections.

4.2 General image-space considerations

These methods are based on the projected area of the enclosing volume. Calculating the projected shape of the convex bounding volume is not trivial, as many special cases arise. This happens when one or two vertices are behind the camera. Negative w-components for the homogenous coordinates of these vertices cause triangle inversions which are computationally intense to handle properly. Degenerate triangles also pose a problem. Some of these problems apply to all two-dimensional methods, whereas others are specific to certain methods. These difficulties are further discussed in chapter 4.8.

4.3 General 3D-space considerations

Constructing a bound for the enclosing volume can also be done in three dimensional space, most notably in view space. The task is to enclose the discs of confusion in a hull. Since non-projected coordinates are used, the graphics hardware will subsequently clip the generated bounding volume and thus all of the problems with the image-space methods never occur in the first place. There are other minor issues regarding near and far clipping that can arise using three-dimensional methods, as will be discussed later, but these methods are generally more robust and require no scene object validity checks.

4.4 Hexagonal bounding volumes

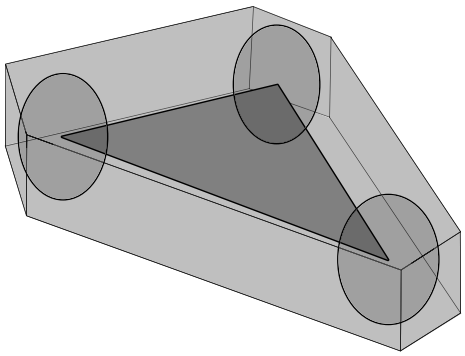

Figure 7: *Extrusion of the bounding hexagon from the triangle plane to 3D. The flat hexagon is inflated along the contained triangle's normal until it tangents the largest disc of confusion.*

For heavily blurred scenes, the best of our proposed bounding volumes is the *hexagonal bounding volume*, shown in Figures 8 and 7. In the 3D version, it is calculated in the following manner:

1. The discs of confusion at the three vertices are projected to the triangle plane.
2. Each of the three edges of the triangle is moved so that it tangents the outside of the larger of the two projected discs at the edge endpoints.
3. At each vertex, a new edge is created with the average angle of the two edges meeting at that vertex. The new edge is then placed as to tangent the outside of the projected disc of confusion at that vertex.
4. The intersection points between the six edges are calculated, forming a flat hexagon.
5. The hexagon is inflated along the enclosed triangle's normal, such that it tangents the largest of the non-projected discs of confusion.

The 2D version is very similar; instead of projecting to the triangle plane, all coordinates are projected to image space, and no inflating is performed at the end.

The resulting hull for the 2D version consists of 6 vertices forming 4 triangles in 1 strip. For the 3D version it is 12 vertices forming 20 triangles in 1 strip. If triangles are considered as one-sided (only

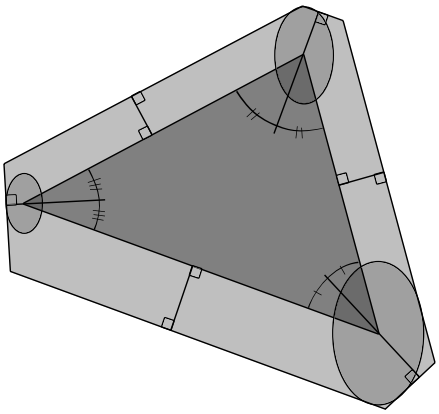

Figure 8: *Bounding hexagon in the triangle plane. The enclosed triangle is depicted in dark grey. The discs of confusion at each vertex of the enclosed triangle, projected to the triangle plane, are shown as transparent ellipses. The generated hexagon is shown in light grey.*

visible when front-facing), this can be reduced to 16 triangles; if the inside of the bounding volume is rasterized, then stochastic sampling on the bounding volume lid will always produce backfacing hits, or similarly for the bounding volume bottom if the outside is rasterized.

The worst case fit for the hexagonal bounding volumes is a nearly degenerate triangle with an extremely large blur radius to triangle size ratio. In this case, the result will mostly resemble a rectangle for the 2D version or a box for the 3D version. The best case fit is when the blur radius is zero. In this case, the bounding volume will exactly fit the enclosed triangle. The hexagonal bounding volumes always produce the most tight-fitting bounds of our proposed methods. However, they are also the most bandwidth-consuming methods and are rather computationally intense. For low-blur scenes, it might therefore be better to use another method since the bound tightness will be less crucial.

4.5 Bounding wedge and parallel edge bounding triangle

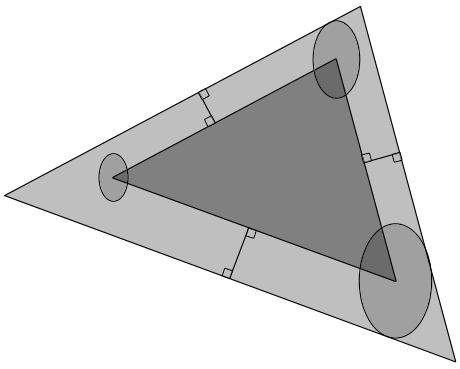

Figure 9: *Bounding wedge in the triangle plane. The enclosed triangle is depicted in dark grey. The discs of confusion at each vertex of the enclosed triangle, projected to the triangle plane, are shown as transparent ellipses. The generated triangle is shown in light grey.*

If the workload on the geometry shader using hexagonal bounding volumes is so high as to make the geometry shader a bottleneck, it is better to produce slightly worse-fitting bounds at reduced computational cost. The *bounding wedge* is essentially the same as a 3D hexagonal bounding volume, but without introducing the averaged edges. This is illustrated in Figure 9. The 2D-equivalent is called the *parallel*

edge bounding triangle, since the bound's edges are parallel to the enclosed triangle's edges.

The best case for the bounding wedge is when the enclosed triangle is entirely in focus, and the radii are zero. The wedge will then exactly match the contained triangle. The worst case is when a triangle is nearly degenerate, in which case a very long bar with a rectangular cross-section is produced. In practice, geometry is modeled as not to have any nearly degenerate triangles, but procedures such as morphing level of detail produce such shards. For morphing objects, it is therefore better to use the hexagonal bounding methods or an adaptive method.

The parallel edge bounding triangle consists of 3 vertices forming a single triangle. The bounding wedge consists of 6 vertices forming 8 triangles in 1 strip. If triangles are considered one-sided, this can be reduced to 7 triangles.

We have developed a reduction technique that generates fewer triangles, covering the same screen area as the full wedge. The technique does contour determination by detecting which sides of the wedge are front facing. The reduction technique uses the same vertices as the original wedge, but connects them differently. This *reduced bounding wedge* method outputs at most 4 triangles in a single triangle strip. The reduced bounding wedge only works if the camera lens is not penetrating the triangle. A downside with our reduction method is that it requires several branches. Contemporary GPUs have poor performance when branching and this increase the strain on the geometry shader.

4.6 Variable radius triangle bound

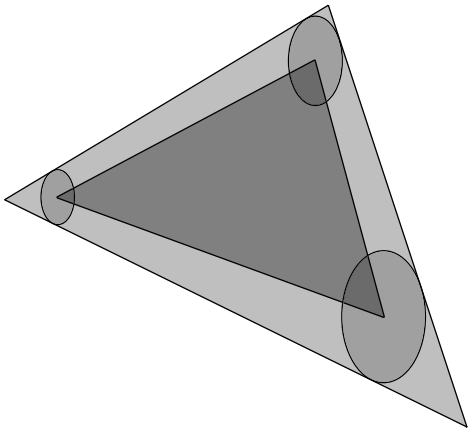

Figure 10: *Variable radius bounding triangle in image space. The enclosed triangle is depicted in dark grey. The circles of confusion at each vertex of the enclosed triangle are shown as transparent ellipses. The generated triangle is shown in light grey.*

As convenient as it might be to create new edges parallel to the enclosed triangle's edges, it is naturally not necessary. Many triangles get a better fit if the new edges are created as to tangent each disc of confusion at the respective end, as shown in Figure 10.

At first glance, it is tempting to draw the conclusion that this *variable radius (VR) triangle bound* always produce tighter bounds than the previously discussed parallel edge (PE) triangle bound. This is not the case. It is easy to construct a case where the area of the VR triangle is significantly larger than for a corresponding PE triangle, and there are cases when it is impossible to create a VR triangle at all.

Also, the difference between a VR and a PE triangle is most notable when there is a large difference of radii between the vertices. The difference in radius is generally small within a single triangle, unless it is extremely close to the camera. The reduced rasterized area does not always yield a workload reduction large enough to compensate the increased computational effort calculating the VR bounds.

There are several problems calculating variable radius triangles. There are quite a few special cases where

the VR algoritm breaks down unless they are properly handled; when calculating the edges tangenting the circles in the variable radius triangle bound, problems arise when no such line exists. These is, for instance, no line that tangents two circles if one is entirely enclosed within the other. There are also inversion problems arising when two edges diverge, caused by a very large disc of confusion radius compared to triangle edge lengths. Handling special cases always adds to the code complexity. It is only worthwhile handling these cases if the even heavier computations still outperform the parallel edge triangle.

These special cases make the variable radius triangle rather unreliable. It is therefore not a recommended technique.

4.7 Adaptive method switching

An application implementing our depth of field effects may choose to apply a single bounding method throughout the entire rendering process. Though this would be easily implemented, performance can be increased by adaptively switching between several techniques. As can be seen in Figure 18, simpler shapes perform far better for low-blur scenes, while heavily blurred scenes benefit from the more complex bounding volumes.

The decision making can be split into two levels; microscopically, the geometry shader can determine which bounding shape to use for each triangle of a mesh. Macroscopically, the application can do an assessment for each object, selecting an appropriate geometry shader mesh-wise. High level decisions can be based on object bounds that usually already exist in any rendering engine for culling purposes. This allows an application to reduce computation and bandwidth costs for the geometry shader in parts where there is very little difference in rasterized area between different methods.

An application might also choose to render objects that are very near the focus plane without the depth of field effect, but this is not recommended since the transition is very hard to conceal.

4.8 Special cases

Constructing bounding volumes and areas is not a trivial task. Care must be taken to utilize them in a correct way, as there are special cases when some of our methods will fail. In this section we will present these and assess the severity and likelihood of each.

4.8.1 Inverted planes

The most severe problem from which all image space methods suffer arises when one or two vertices are located behind the camera. The problem that occurs can be seen in Figure 11 where one vertex travels in the camera view-direction from the positive to the negative domain. The projected point wraps around infinity and re-enters the screen area from the opposite direction with a negative homogenous w-component. Blindly creating bounding areas based on the projected xy-coordinates clearly results in invalid bounds. This is a severe problem, limiting the use of these methods to objects entirely in front of the camera. Any object reaching behind the camera must use a volumetric method working in 3D space since they inherently avoid such problems as they do not use projected points in the first place.

4.8.2 Near and far clipping

While the volumetric methods do not suffer from plane inversion problems, they instead have the problem of bounding volumes extending into a clipping plane even if the contained triangle does not. This may or may not affect rendering; if the outside of the bounds is rasterized and the rear of the bounding volume is clipped by the far clipping plane, the resulting rasterized surface will still cover the exact bounding area. If the inside of the same volume is rasterized instead of the outside, the clipped region will erroneously not be processed by the fragment shader. The opposite is true when the bounding volume intersects the near clipping plane; if the outside of the volume is rasterized, the clipped region will not be rendered. If the inside of the volume is rasterized, it will not be affected by removal of the clipped parts of the

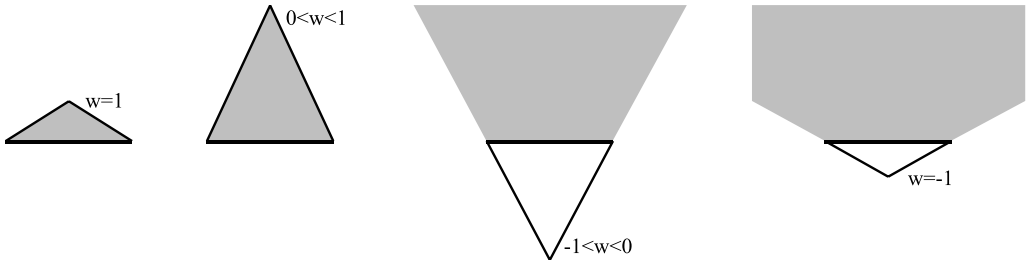

Figure 11: *Image space triangle inversion for negative w-components. View space x and y coordinates are identical for the four images.*

volume. For interactive applications, we recommend rasterizing the inside of the bounding volumes due to better resilience to camera penetration.

The solution presented above can only be guaranteed to work if the contained triangle itself does not intersect a clipping plane. In applications not relying on near and far clipping planes in order to get correct rendering, this is not a problem. If geometry is rendered very close to the near clipping plane, it is recommended to rasterize the insides of the bounding volumes. Geometry known to always be distant may be safer to render using the outside of the bounding volumes.

The image-space methods always output the generated bounding areas in the same depth plane, effectively disabling near and far clipping problems.

The reduced wedge triangulation does not unambiguously rasterize the outside or inside of the bounding volume, but rather whatever polygon happens to connect the points along the silhouette. It is therefore impossible to predict the outcome of clip plane intersection. If such intersections are expected this method is inappropriate.

4.8.3 Degeneration

If a triangle is degenerate, our methods result in undefined behavior. In order to safely handle this with minimal computation cost the geometry shader should detect and discard any degenerate triangle.

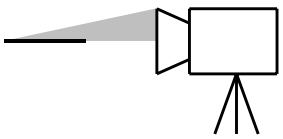

Figure 12: *Surfaces degenerate in image space are still visible due to the non-zero radius camera lens.*

When a non-degenerate triangle becomes degenerate in image space it will still be visible from some parts of the camera lens, seen in Figure 12. The image space methods will fail to construct correct bounds for triangles degenerate in image space, resulting in undefined behavior. Our proposed way of correctly handling these cases would be to detect zero-area triangles and generating object aligned bounding rectangles instead.

5 Stochastic pixel shading

In this chapter, our stochastic sampling pixel shader is explained. Finally, we will discuss some details that deserve further attention.

5.1 The novel algorithm

When the bounding geometry generated in the geometry shader gets rasterized by fixed function hardware, the rasterizer interpolates all vertex parameters linearly in homogenous space. Since each vertex of the bounding volume contain the same parameters — except for position — they will be identical for the entire rasterized area around the contained triangle.

The data that is supplied to the pixel shader is the following:

- The position of each of the three corners of the enclosed triangle
- The signed disc of confusion radius at each of the three corners
- Any additional vertex parameters needed for fragment processing, e.g.:
  - Normal at each corner
  - Texture coordinates at each corner
  - Diffuse color at each corner, etc.

The first task for the pixel shader is to stochastically perturb the enclosed triangle. This is done by randomizing a perturbation vector uniformly on the unit disc in the XY-plane and offsetting the corner positions by this vector multiplied by the respective disc of confusion radius.

After randomly perturbing the enclosed triangle, an intersection test has to be done with the ray going through the fragment center and the pinhole camera. This is the pinhole camera eye vector, here denoted $eye_{pinhole}$. The barycentric weights e_i for the intersection point with the triangle plane are calculated using the vector $eye_{pinhole}$ and the perturbed vertices P_i as follows:

$$e_i = (P_{i+1} \times P_{i+2}) \cdot eye_{pinhole}, \quad i = \{0, 1, 2\}$$

These three dot products may be evaluated as a single matrix-vector multiplication.

If any component of the barycentric weights is negative, the ray does not intersect the triangle. In that case, the fragment is discarded. This will happen in the semi-transparent region of the triangle, show in Figure 5. In the areas covered by the bounding area that are outside the exact bounding area, all fragments will be discarded.

If there is an intersection with the triangle, execution continues and the barycentric weights are normalized to form the barycentric coordinates:

$$e_{sum} = e_0 + e_1 + e_2, \qquad u = \frac{e_0}{e_{sum}}, \qquad v = \frac{e_1}{e_{sum}}, \qquad w = \frac{e_2}{e_{sum}}$$

The depth and the additional shading parameters are then interpolated using these barycentric coordinates as weights. The intersection position can be interpolated from the enclosed triangle vertex positions. After this, the usual pixel shading takes place using the newly calculated parameter values.

Lighting calculations for a pinhole camera use an *eye vector*, originating from the point being shaded, directed towards the pinhole camera. If the lens has a non-zero radius, the light path variations has to be taken into account. We introduce a *lens vector* eye_{lens} originating from the point being shaded, directed towards the point on the lens from which the scene is viewed for the current perturbation.

It is straightforward to construct the lens vector by calculating the perturbed eye position on the lens and subtracting it from the interpolated intersection point. It is important to note that eye_{lens} differs from $eye_{pinhole}$, of which the latter should not be used in lighting calculations.

When fragment shading is done, the interpolated depth is returned along with the computed color.

5.2 Tie-breaker rules

If an edge between two triangles cross exactly through a pixel center, the pixel shader clipping rules described above will pass fragments from both triangles. Rendering the same fragment for both triangles will not yield any visual artifacts if both triangles are opaque and the shading across the edge is continuous. If they are not opaque, rendering two fragments will result in incorrect blending.

To avoid such artifacts, it is possible to create a set of tie-breaker rules. Such rules generally have several comparisons that can be simultaneously performed very efficiently in hardware, but are rather time consuming to perform in a pixel shader. Tie-breaker rules should thus only be used if the artifacts are truly visible and are found disturbing.

5.3 Random perturbations

For our perturbations we need two (pseudo-)random numbers. Our method of choice is to sample a texture filled with random numbers pre-calculated on the CPU.

There are several advantages using the pre-calculated texture:

- Only a single texture lookup is required for the fragment shader to obtain the two random numbers that are needed.
- The random numbers can be stratified in advance.
- Any aperture shape can be produced by limiting the random vectors in the texture to a subset of the unit disc. Time-varying apertures can be simulated by making the distribution density proportional to the local exposure time.

A drawback with the sampling method is that each fragment requires an additional texture lookup. However, the texture coordinate is simply the screen coordinate of the pixel, scaled to a one-to-one pixel-to-texel ratio; the continuity is therefore very good and the texture cache should be effective when performing this lookup. Aside from the increased texture bandwidth, performing such a texture lookup is highly efficient on typical hardware. Impacts on caching are further discussed in chapter 9.3.

5.4 Depth culling

The algorithm above calculates the depth of each fragment; this means that the fixed function hardware cannot discard regions that are outside the valid depth range $[0, 1]$ before the pixel shader is evaluated. If the application renders a significant amount of geometry outside these bounds, it may be advantageous to perform a clipping operation on the depth value as soon as it has been interpolated, thus avoiding the fragment lighting calculations.

It might also be more efficient to perform depth culling in the geometry shader since the entire depth interval is known at that stage.

6 Integrating other effects

It is crucial that any technique should be compatible with other visual effects. This chapter describes how other shading effects can be integrated with our technique.

- Vertex effects

  Many techniques such as skinning, morphing, space warping, wave generation and texture coordinate generation operate solely on the vertex shader. As long as their output isn't infinite points with zero homogenous w-components, they can simply be prepended to our vertex shader.

  Since our geometry shaders works in view space, a requirement of the other vertex shader effects is that any output must be able to be converted into view space. It is possible to rewrite our geometry shaders to work in projection space, however that would further increase computational complexity.

- Geometry effects

  Geometry shaders can produce data dramatically different than the input, and are thus difficult to generalize.

  - The easiest way to handle data amplification and extensive modification of input data is to move the calculation of the DoC radius from the vertex shader to the beginning of our geometry shaders, and executing the bounding shader sequentially on each output triangle of the integrated geometry effect.
  - A multi-pass approach is also possible: in a first pass, vertex and geometry shading with other effects is performed, and the output is stored to a vertex buffer. A subsequent pass is then performed on this vertex buffer using our shaders.

- Pixel effects

  Almost any pixel shader can be appended to our stochastic pixel shading. The main limitation is the number of interpolated registers used. Every vertex attribute needs three registers, and the maximum allowed register count is thus reached much sooner.

7 Performance and quality considerations

In this chapter, various important performance issues are discussed.

7.1 Bottlenecks

There are mainly three points in the graphics pipeline which receive heavy load during stochastic rasterization. The first point is the geometry shader, where the bounds are calculated. The more complex the bounding volume, the more work is required from the geometry shader.

The second potential bottleneck in the pipeline is the input stage to the fixed function rasterizer. Since our geometry shaders do moderate to heavy data amplification, the bandwidth between the geometry shader and fixed function rasterizer may become a limitation.

The major performance bottleneck is the pixel shader. First, intersection testing and parameter interpolation are now done in the pixel shader – this is normally handled by the fixed function rasterizer. Second, super sampling vastly increases the rasterized surface compared to the final image. Third, there is added overdraw due to the padding of each triangle.

In order to balance these weak links optimally, the choice of bounding solution should be made based on the circle of confusion radius in pixels.

7.2 Perceptive considerations

Rendering with the depth of field effect increase the work of calculating the final image. Since the main visual impact of the effect is actually detail degradation, this is not always necessary.

Mesh reduction is the key to achieve good performance using SR. The main performance hit compared to non-depth-of-field rendering is a result of massive overdraw that is later discarded due to the padding of every triangle in the scene. The amount of overdraw is highly dependent on the blur radius compared to the triangle size. It is therefore crucial to have a good mesh reduction system to prevent rendering tiny triangles that will never be resolved due to blur.

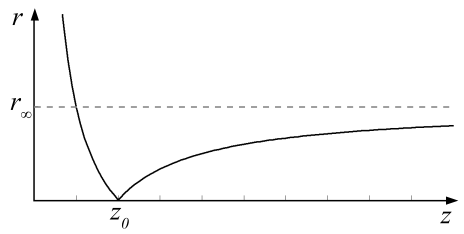

Figure 13: *Circle of confusion radius vs. object distance. As z grows towards infinity, the radius is bound by r_∞.*

Traditionally object detail has been dynamically reduced as the object distance increases. This avoids rendering details on objects far away that would not be resolved. The smallest resolved detail is proportional to r/z, where r is the resolution and z the object distance. With depth of field, this quality reduction can be drastically extended since the circle of confusion radius is inversely proportional to z for objects closer than the focus depth, as seen in Figure 13. Objects close to the camera have previously required the highest model detail. Unless they are in focus, they now become less resolved and can thus be rendered with reduced geometric quality, demonstrated in Figure 14. For objects far away, detail visibility is still inversely proportional to object distance with some higher attenuation factor.

Latest hardware technology have taken a turn to provide mesh tesselation as an alternate means of detail control. The same guidelines are valid for mesh tesselation as with mesh reduction; detail level should be chosen not only based on distance, but also on blur radius.

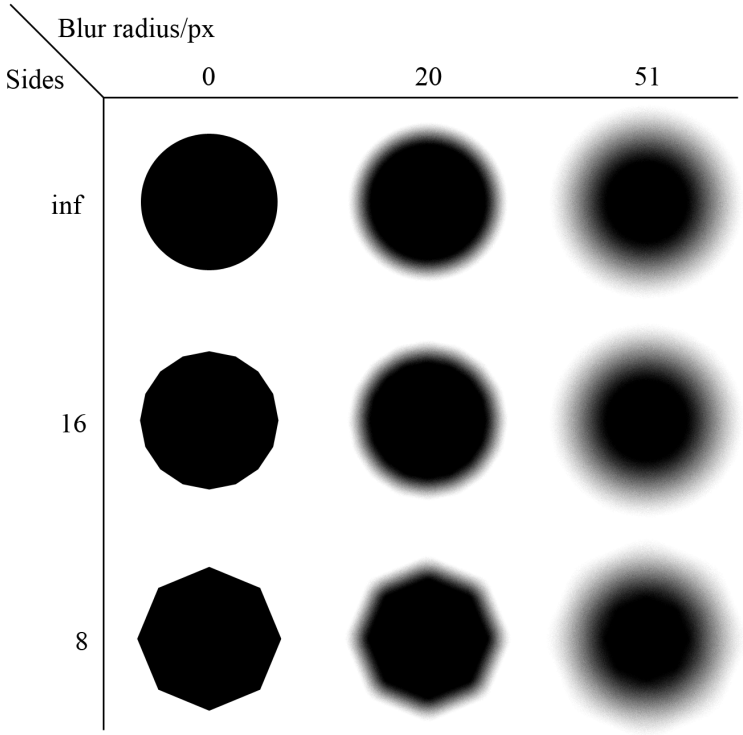

Figure 14: *Detail reduction impact with variable circle of confusion radius.*

It is important to note that mesh reduction in the foreground affects large screen areas; unless the method used is morphing between quality levels, severe popping will be visible despite heavy blur.

8 Comparisons to accumulation buffer techniques

8.1 Image quality comparisons

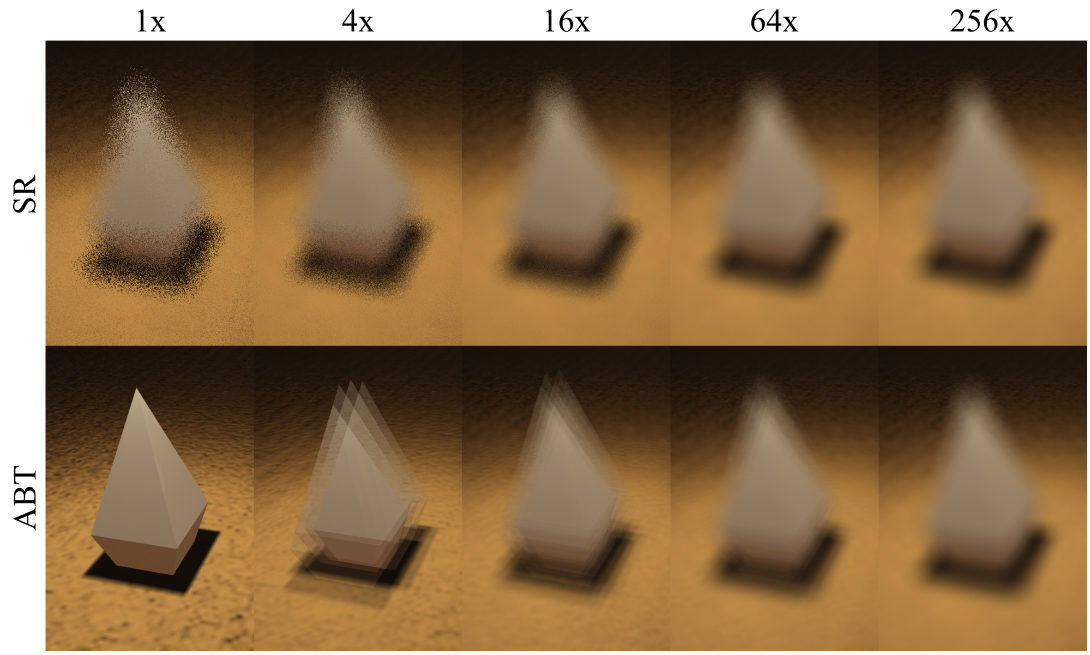

Figure 15: *Image quality comparison between stochastic rasterization and accumulation buffering at different sampling rates.*

Figure 15 shows a pylon at different quality levels using stochastic rasterization and accumulation buffering. Areas of interest include the top of the pylon, the ground, and the shadow at the pylon base.

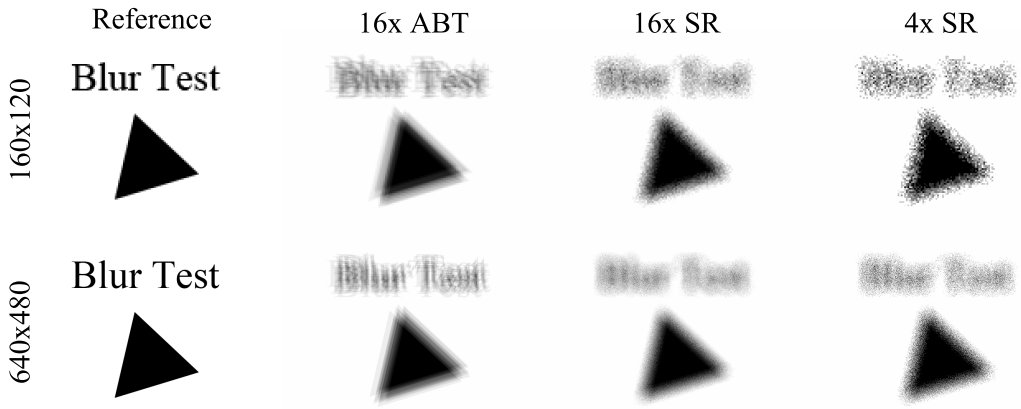

Figure 16: *Image quality comparison between stochastic rasterization and accumulation buffering at different resolutions.*

Figure 16 shows a triangle and the text *Blur Test* at two resolutions using different techniques and sampling densities. Increasing the resolution using accumulation buffering does not increase image quality; as the resolution increase, ghosting gets more pronounced. The image quality of stochastic rasterization increases with the resolution.

8.2 Performance comparisons

All tests were performed on a PC equipped with an Intel Dual Core D945 3.4GHz CPU, 2GB RAM and a GeForce8800 GTX with 768MB VRAM. The tests were run in a resolution of 1280x720.

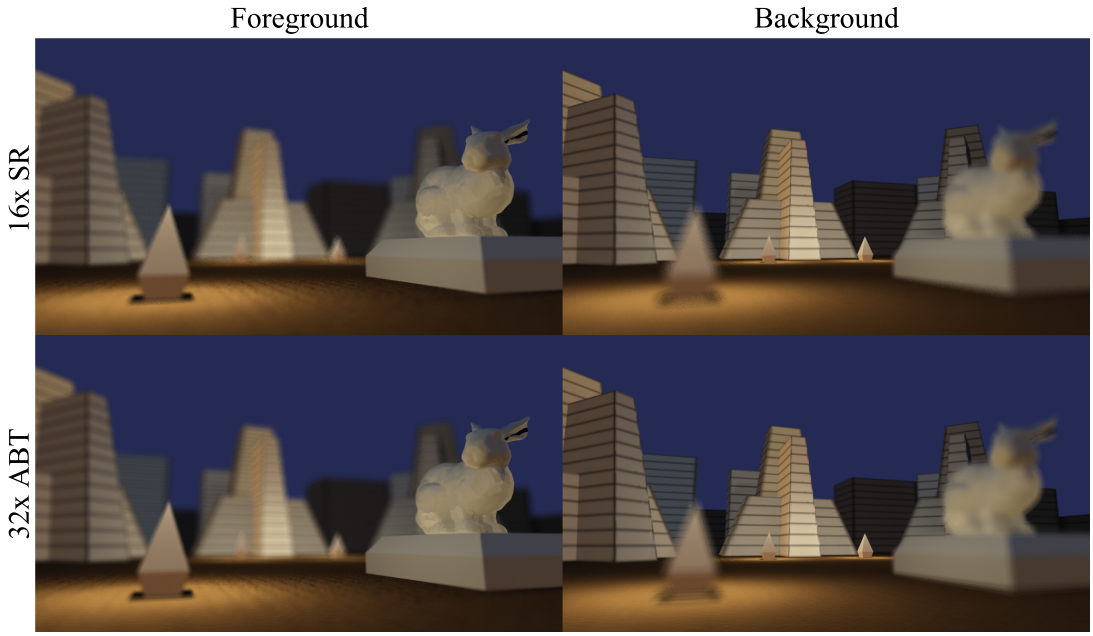

Figure 17: *First and last frame of focus sweep from foreground to background for stochastic rasterization and accumulation buffering.*

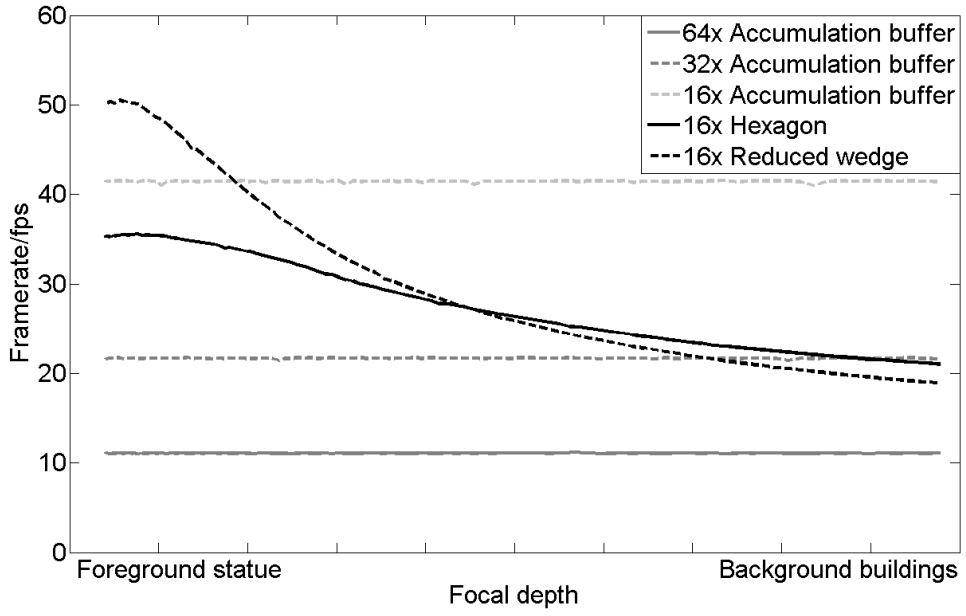

Figure 18: *Performance during focus sweep from foreground to background for different methods. Black lines are stochastic rasterization. Medium grey lines are approximately of equal quality using accumulation buffering. The light grey line is equal sampling rate using accumulation buffering.*

Figures 17 and 18 show a focus sweep from a statue in the foreground to some distant buildings. Since perceived quality is highly subjective, both 32x and 64x accumulation buffering are shown in the plot. The 16x accumulation buffering plot shows overhead induced performance loss compared to the reduced wedge when the foreground is in focus.

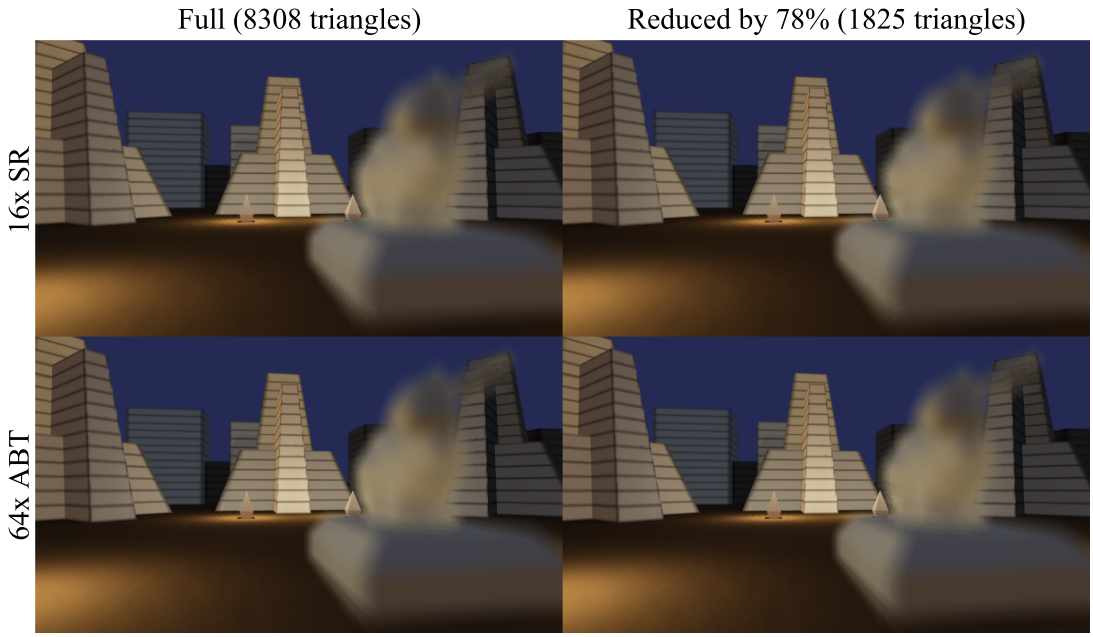

Figure 19: *Visual impact of geometric reduction, shown for stochastic rasterization and accumulation buffering. Only the bunny statue is reduced.*

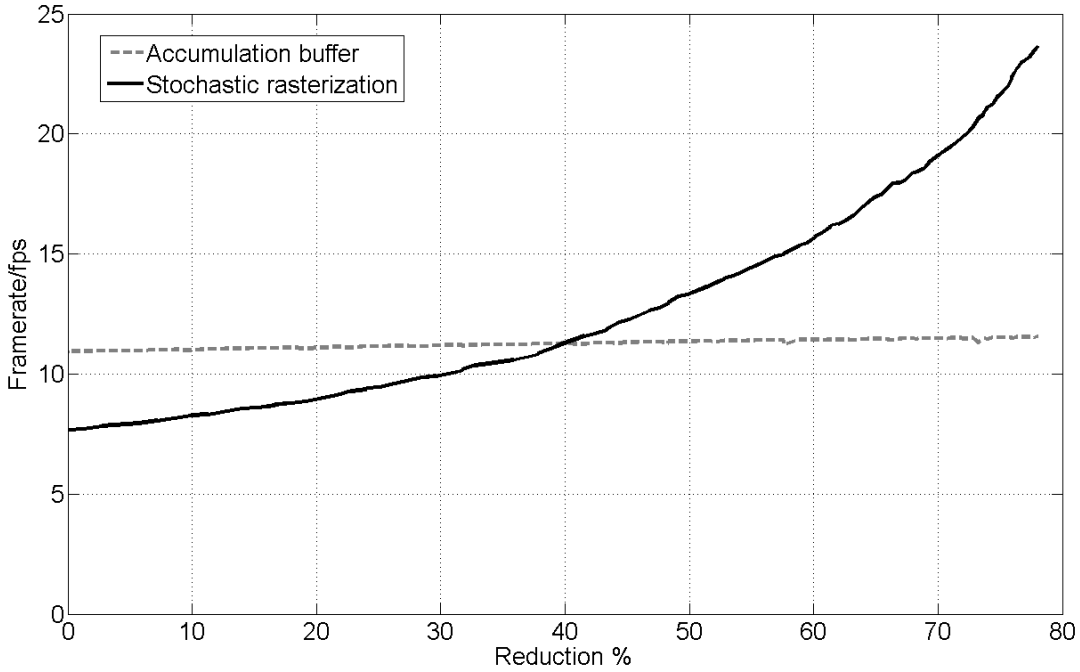

Figure 20: *Framerate impact of geometric reduction, shown for stochastic rasterization (16x) and accumulation buffering (64x). As reduction increases, performance becomes real-time.*

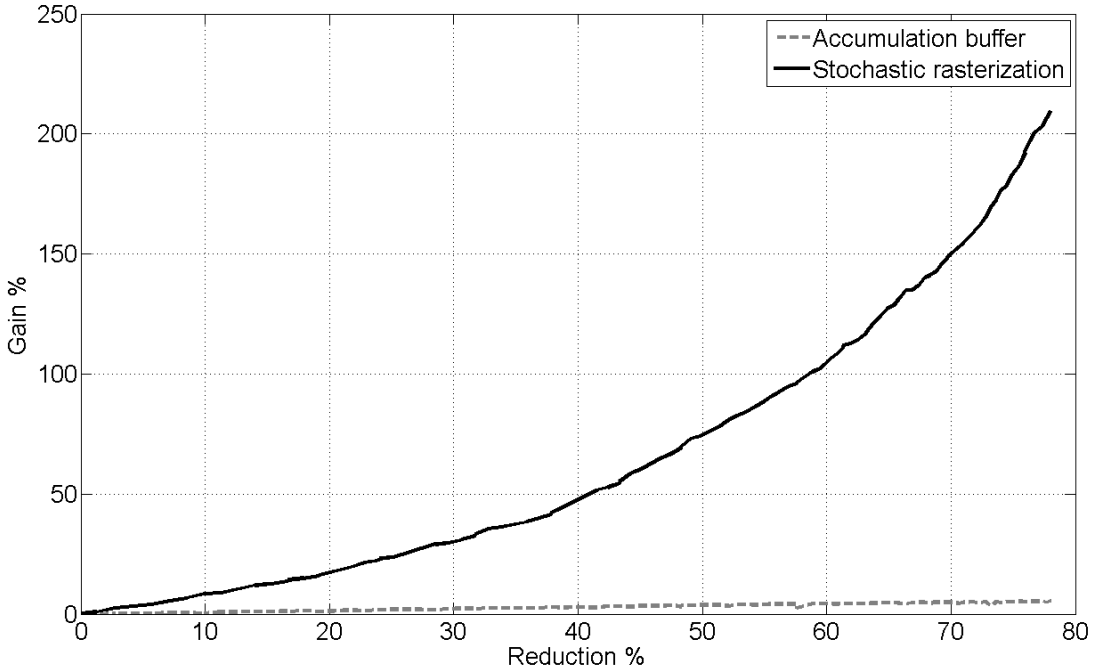

Figure 21: *Performance gain by geometric reduction, shown for stochastic rasterization (16x) and accumulation buffering (64x).*

Figures 19, 20 and 21 illustrate various aspects of geometric quality reduction, in this case in the foreground. Reduction is solely performed on the bunny statue.

9 Discussion

We have shown that stochastic rasterization is possible at quality and speed comparable to current accumulation buffering techniques. We have developed a fairly well optimized framework on which all our tests were performed. Our results and various aspects of our method are discussed in the following subsections.

9.1 Results

All perceived image quality comparisons are subjective [Richardson 2003], and the conclusions drawn here are based upon the authors' judgements.

Figure 15 clearly shows that equal sampling rates does not mean equal quality. We find that approximately four times as many samples are needed for accumulation buffering to achieve a comparable result. In the lowest quality regions, this number increases somewhat. Depth of field is about directing focus; objects outside the focal range are not supposed to draw attention. Noise does in general not draw as much attention as ghosting and is therefore more desirable.

Figure 16 shows the effect of increasing resolution. The circle of confusion radius is kept fixed in physical units, which means equally magnifying the circle of confusion in pixels. For accumulation buffering, no improvement is achieved by increasing resolution; the pixellation is reduced at the cost of increased ghosting. For stochastic rasterization this is not the case; pixellation is reduced and the noise gets more dense and thus less pronounced.

Performance scales with resolution only if the rendering speed is fillrate bound. For stochastic rasterization, this is the case most of the time. This means that while SR gains quality by increasing resolution, it also loses performance. Accumulation buffering, on the other hand, is seldom solely fillrate bound and does therefore not suffer an equally large performance penalty. This performance loss for SR is of the same magnitude as the needed increase in sampling density for ABT to achieve the same quality increase. It is easily realized that the same argument applies to varying the aperture size.

Figure 18 shows how performance is affected by the ratio of triangles in focus and out of focus. The largest circle of confusion radius is almost constant throughout the sweep, keeping the needed sampling rate fixed. The entire scene except the statue is comprised of just a few hundred triangles, while the statue itself has about 8000 triangles. It is seen that performance is highest when most triangles are in focus. This is because of reduced average overdraw — overdraw is dependent on circle of confusion area compared to the triangle screen area. Also important to note is that all objects within the scene are rendered using the same method. Dynamically choosing the appropriate method (switching between reduced wedge and hexagon per object) would make performance at least equal to the best of the two methods at every focal depth, and even better for some intervals. This would be more pronounced if the triangles were more evenly distributed across the scene, as is most often the case.

The most demanding regions are highly tesselated surfaces that are out of focus. However, the smallest resolvable detail is inversely proportional to the circle of confusion as discussed in chapter 7.2, until the pixel size is reached. The number of triangles needed for an object is thus proportional to the covered screen area divided by the circle of confusion size. The concept is illustrated in Figure 14. Figure 19 shows this effect in practice; it is hard to see any difference between the fully resolved statue and a heavily reduced version.

Combining the reduction argument above with what can be seen in Figure 21, it is obvious that the performance decline visible in Figure 18 can at least be compensated. Note that the performance gain shown in Figure 21 is caused by decreased overdraw and is valid when the rendering speed is fillrate bound. If instead rendering speed is geometry bound, the rendering time is proportional to the number of triangles and the reduction gain is similar.

Accumulation buffering does not benefit from any significant performance boost by geometric reduction if the rendering speed is fillrate bound. For geometry bound renderings, the performance gain is equal

to that of stochastic rasterization. If the rendering speed is CPU bound, then no performance gain is obtainable at all by mesh reduction. It is most often desirable to have a balance between the three bounds, avoiding any bottleneck. There is therefore typically limited increase in performance when reducing geometric detail using accumulation buffering.

9.2 Derivatives

One of the disadvantages of our implementation of SR is that hardware-provided functional derivatives are erroneous: graphics hardware estimate derivatives by first order differences between fragments, but since neighboring fragments have scattered vertex attributes this will provide invalid estimations. Especially, since texture lookups from neighboring fragments will be scattered across the texture surface, automatic miplevel selection will grossly underestimate the appropriate texture resolution. Regular texture lookups based on automatic derivatives will therefore yield smudged surfaces.

It it possible to circumvent the mipmapping errors simply by disabling mipmapping. While resulting in sharp textures, this method has the usual problem with aliasing when texture resolution is greater than the sampling density. Super sampling lessens the aliasing, but does not eliminate it entirely.

A better solution is to calculate the texture coordinate derivatives at the three vertices of the enclosed triangle in the geometry shader. These derivatives will then be interpolated in the pixel shader along with all the other shading parameters. These true derivatives can then be supplied to the texture sampler, allowing proper miplevel selection.

Interpolating the texture derivatives has the obvious disadvantage of requiring additional registers. In return, the visual quality should actually become better than using the built-in hardware first order difference estimation since the derivatives at the sample centers are obtained.

9.3 Texture cache

Stochastic rasterization will result in high sampling strides due to incoherent sampling patterns. This should reduce texture cache efficiency if the locally sampled texture coordinates span over a larger area than can be fit into the texture cache. Sampling multiple textures reduce the area that can be accommodated in the cache for each texture, as they have to share the available space. In our solution, a part of the cache will be occupied by the sample distribution texture, reducing the capacity for other textures. The sampling pattern for the sample distribution texture is highly spatially coherent, so it should only occupy small amounts of the cache. The spatial sampling range of other textures depend mainly on the circle of confusion radius of the textured surfaces.

9.4 Super sampling and stratified distributions

Super sampling is accomplished by rendering to an oversized render target, and subsequently re-scaling the render target to the actual resolution. For instance, a 1024x768 image might be rendered to a 4096x3072 render target, and each pixel of the final image will be the average of 4x4 samples in the render target.

If each sample is randomly distributed, this method produces unbiased images. Introducing stratified sampling may bias the image unless proper care is taken. In order to avoid biasing, the geometry perturbation parameters must be uncorrelated to the sample position within a single super sampling block. This might be accomplished by first generating the stratified offsets and then randomly permuting them within the super sampled pixel block.

It is not necessary to generate unique random sample offsets for each and every pixel of the supersized render target. It is sufficient to create a block of about 512-by-512 samples or even less and repeat this pattern across the entire render target. The screen space tile size is reduced if super sampling density is increased, but the repeating pattern gets less pronounced due to the increased quality.

High quality super sampling requires a lot of memory and large texture sizes. This might prevent a direct rendering of an image in one pass; for instance, DirectX 10 has a specified maximum texture size of 8192-by-8192 pixels. If a larger supersized render target is required, the image has to be tiled. This is actually no real limitation; if resolutions are that great, it is highly probable that the pixel shader is the only bottleneck in the rendering and the additional triangle setup work caused by the tiling process will not impair performance.

9.5 Deferred shading

Typical graphics hardware work on two-by-two-pixel blocks in order to produce derivative estimations. If any one of four pixels needs to evaluate a code path, all four pixels will do so. Those pixels that did not intend to evaluate that path simply discard the results. During normal rasterization, this has little effect since only a few pixels at each edge will be evaluated in vain. SR will have large amounts of blocks with only one or two live pixels. In order to avoid wasted computational power, the code path after evaluating whether a pixel is inside the triangle or not should be as short as possible.

One way of reducing unnecessary evaluations is using deferred shading, with one deferred pass before each state change: As soon as the triangle attributes have been interpolated in the regular draw calls, they can be stored in render targets. A stencil buffer should be used to mask the areas that were written to during these draw-calls. The deferred pass should then render a single full-screen quad, only operating on areas masked in the stencil buffer. This pass should read the interpolated values and perform the actual shading.

Which state changes that require a deferred pass is not unambiguous. If textures are changed very frequently it might be more efficient to store an index in the pre-pass and using it to index a texture array in the deferred shading pass. Likewise, it might be reasonable to store a small amount of frequently changing shader constants to render targets in order to reduce the total amount of deferred shading passes. Performance gain should be highest if shaders, textures and shader constants are changed as infrequently as possible. However, if one of these are changed, all of them may be changed at no additional cost.

Using deferred shading with SR will reduce the number of shading evaluations, but it will not reduce the number of interpolation evaluations. Also, it uses up lots of memory and bandwidth. It is therefore probably only worthwhile if the shading procedure is complex.

9.6 Suggested hardware improvement: triangle- and strip constants

One of the major issues with SR is the large amount of vertex attributes that need to be passed to the pixel shader. Most of that data is constant for the entire bounding geometry. For instance, if the desired effect is a bump mapped textured surface with depth of field, then each vertex in the bounding geometry would use a minimum of 39 floats (bound: position, contained triangle: position, normal, tangent and texcoord for each of the three vertices). Of these, only 4 floats are varying (bound position), while the other 35 are constant. These 35 constant floats are copied to each of the output vertices on the bounding geometry. Introducing *triangle constants* would have two major benefits:

- Reduced bandwidth from the geometry shader
- Reduced utilization of interpolation units

The bandwidth gain is most pronounced for triangle lists, and strips of only a few triangles. If long triangle strips are used, the bandwidth gain is neglectable. However, the same concept can also be expanded to *strip constants*, which would be constant within all triangles of a single strip. This would further reduce bandwidth from the geometry shader, especially for longer strips. The impact of our suggestion is shown for the above described example in table 2.

Other applications would doubtlessly also benefit from this kind of constant banks, albeit not as much as SR. Applications could for instance make larger draw-calls by moving frequently varying pixel shader

Bounding geometry type	Constant type	Total output floats	Interpolated floats per triangle
3D hexagon	None	858	39
	Triangle	788	4
	Strip	123	4
2D hexagon	None	234	39
	Triangle	164	4
	Strip	59	4

Table 2: *Impact of triangle and strip constants on bump mapped, textured surface with depth of field*

constants to strip constants. This would not only reduce the number of draw calls but also reduce constant modification calls, resulting in less overhead.

9.7 Suggested hardware improvement: depth-interval

Modern hardware tiles the depth buffer and keeps track of the minimum and maximum depth within each tile. This way depth culling can be done quickly for an entire tile; if a triangle to be rendered is farther away than the tile maximum, then all fragments can be discarded at once. If instead the entire triangle is closer than the minimum depth value within a tile, then no depth comparison is needed pixel-wise since it is already known that all fragments will pass the depth test. Not only do these optimizations reduce the number of depth comparisons, but they also reduce bandwidth usage to the depth buffer.

If a fragment shader writes to the position register, and thus makes the depth of each fragment unpredictable, the depth optimizations are disabled on current hardware. Performance is reduced even further since the depth test has to be performed after fragment shading, resulting in wasted computations. The nature of stochastic rasterization requires writing to the position register since the depth of each fragment is interpolated in the fragment shader. Thus, SR cannot take advantage of depth optimizations.

Even though the depth of each fragment is unknown beforehand, a conservative interval may easily be provided on several levels:

- Bounds for an entire object can be calculated by the application. This should suffice to perform optimized inter-object depth culling.
- Bounds for each vertex can be calculated by the vertex shader. These bounds can be interpolated across each triangle, forming interval planes. Alternatively, the minimum and maximum of the vertices comprising each triangle can be used.
- Bounds for each triangle can be calculated by the geometry shader. This would benefit from triangle constants for storage as discussed in 9.6.

If the depth buffer is tiled at several resolutions, a combination of object depth-bounds and either vertex or triangle depth-bounds would have the greatest potential, with coarse inter-object culling mainly taken care of by the object depth-bounds while finer intra-object culling may be performed with the vertex or triangle depth bounds.

A more lightweight method of providing at least partial depth optimization is to provide a render state where the application can guarantee that no depth writes will be greater than — or less than — the interpolated depth of the fragment. In the case of SR, where we always rasterize the backside of the bounding volume (as discussed in chapter 4.8.2), this would suffice to take advantage of depth culling and to prevent evaluating the pixel shader for hidden fragments. The advantage of this method is that it only requires very slight hardware modifications.

9.8 Suggested hardware improvement: programmable raster shader

It is an undeniable trend that more and more of the graphics pipeline is becoming programmable. The rasterizer is still fixed function, and it might be of interest to consider what could be achieved by making it a programmable shader unit. The main task of such a *raster shader (RS)* would be to take a single triangle input and produce a set of fragments that are overlapped by the input triangle. The vertex attributes should be interpolated for each of the fragments in the output fragment set. Should triangle and strip constants be implemented, as discussed in 9.6, then these should be accessible from the raster shader. It should also preferably have access to shader constants. Ideally, the raster shader should work in some hierarchical manner, performing coarse tile-wise evaluations at first and gradually refining the evaluation to smaller regions, much as current fixed-function rasterization works.

Using such a raster shader, the stochastic perturbation and intersection test could be performed before the fragment shader is ever evaluated. This would allow for much better subsequent mapping of the jittered live fragments to the SIMD units, thus avoiding much of the unnecessary work. This would give the same advantages as deferred shading, as discussed in section 9.5, but without any of the drawbacks. No additional memory would be needed, no deferred pass would be performed, and the only modification to the application render loop would be to load the raster shader prior to rendering.

The fragment to SIMD unit allocation process would be vastly improved if the two-by-two block restriction was lifted; it would be a simple task for the device driver to properly detect whether a particular shader needs access to derivatives and only activating blocking if needed. Normally derivatives are used for texture filtering, but in SR that is not the case. If no hardware-supplied derivatives are accessed, fragments become independent of each other and arbitrary mapping to SIMD units is possible.

Also, the depth-interval culling discussed in 9.7 could be implemented with ease using a raster shader.

Other impacts of a programmable raster unit include the ability to perform sub-triangle culling; a PCU [Hasselgren and Akenine-Möller 2007] could be implemented on the raster shader, reducing the sensitivity to bounding geometry tightness. Also, rasterization of higher order primitives, as discussed in section 9.9, would gain much being able to evaluate functions hierarchically.

9.9 Other uses of our work

A new possibility that comes with our technique is the ability to sample non-linear surfaces. Since the intersection testing and parameter interpolation is done in the pixel shader, there is no inherent limitation to triangle based geometry. Any shape can be used, as long as a conservative bound can be calculated in the geometry shader. Using higher order primitives might be a better alternative than doing tesselation, since it results in better quality and less overdraw due to padding. Using higher order primitives will naturally be slower in areas that are in focus, but in blurry regions it has great quality and performance potential. This tradeoff might be worthwhile in many applications, since the blurry regions in general are the more computationally intense areas. Any modification that ease the workload on blurry regions at the cost of higher workload on sharp regions result in more fluid framerates, since the magnitude of blur then has less impact on performance.

While in theory any shape can be chosen, such as cylinders and spheres, it is probably only worth implementing flexible shapes, for instance Bézier-surfaces, capable of modeling generic surfaces since too specialized shapes would break large draw-calls.

Another potential use of our technique is rendering single-pass glossy reflections. An interesting application that has only been feasible using ray tracing until now is per pixel variable glossiness.

References

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic Rasterization using Time-Continuous Triangles. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, 7–16.

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed Ray Tracing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, 137–145.

DEMERS, J. 2004. Depth of Field: A Survey of Techniques. *GPU Gems*, 375–390.

HAEBERLI, P., AND AKELEY, K. 1990. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)*, 309–318.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2007. PCU: The Programmable Culling Unit. *ACM Trans. Graph. 26*, 3, 92.

NVIDIA. 2006. Microsoft DirectX 10: The Next-Generation Graphics API. Technical brief, November.

RICHARDSON, I. E. 2003. *H.264 and MPEG-4 Video Compression*. Wiley.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-Accelerated High-Quality Hidden Surface Removal. In *Graphics Hardware*, 7–14.