High-Quality Spatio-Temporal Rendering using Semi-Analytical Visibility

Carl Johan Gribel
Lund University

Rasmus Barringer
Lund University

Tomas Akenine-Möller
Lund University and Intel Corporation

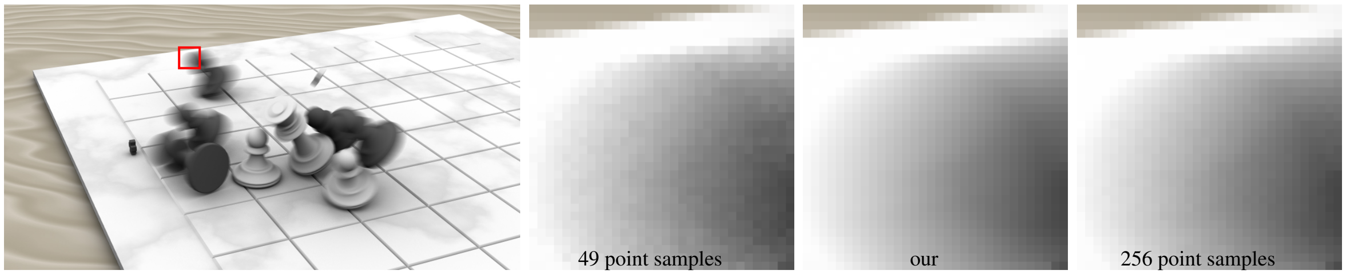

Figure 1: *A chess scene with motion blur rendered with stochastic rasterization with 49 point samples, our semi-analytical visibility algorithm in the temporal domain with four line samples in the spatial domain, and finally with stochastic rasterization with 256 point samples. Our work focuses on spatio-temporal visibility, and for 49 samples it takes* 3.8 *seconds to compute visibility and simple shading (ambient occlusion not included) at* 1024×768 *pixels. With these settings, our algorithm computes the middle image in* 3.6 *seconds. Note that the image with 49 samples is rather noisy, and even with 256 samples, there is still some noise, while the motion in our image is essentially free of noise. Furthermore, the quality of the spatial anti-aliasing (look at the static edge at the top) in our image closely matches that of 256 point samples.*

Abstract

We present a novel visibility algorithm for rendering motion blur with per-pixel anti-aliasing. Our algorithm uses a number of line samples over a rectangular group of pixels, and together with the time dimension, a two-dimensional spatio-temporal visibility problem needs to be solved per line sample. In a coarse culling step, our algorithm first uses a bounding volume hierarchy to rapidly remove geometry that does not overlap with the current line sample. For the remaining triangles, we approximate each triangle's depth function, along the line and along the time dimension, with a number of patch triangles. We resolve for the final color using an analytical visibility algorithm with depth sorting, simple occlusion culling, and clipping. Shading is decoupled from visibility, and we use a shading cache for efficient reuse of shaded values. In our results, we show practically noise-free renderings of motion blur with high-quality spatial anti-aliasing and with competitive rendering times. We also demonstrate that our algorithm, with some adjustments, can be used to accurately compute motion blurred ambient occlusion.

CR Categories: I.3.3 [Picture/Image Generation]: Antialiasing; I.3.7 [Three-Dimensional Graphics and Realism]: Color, shading, shadowing, and texture;

Keywords: analytical visibility, anti-aliasing, ambient occlusion, motion blur

Links: DL PDF

1 Introduction

Visibility computations is a fundamental core research topic in computer graphics, and it has been active and vivid for more than 45 years. Algorithms for visibility play a central role in essentially any type of rendering including, for example, rasterization, ray tracing, two-dimensional graphics, font rendering, shadow generation, global illumination, and volume visualization.

During the 1970's and 1980's, research on analytical visibility for spatial anti-aliasing [Catmull 1978; Weiler and Atherton 1977] and motion blur [Korein and Badler 1983; Catmull 1984; Grant 1985] was rather popular. However, after Cook et al.'s stochastic point sampling approaches were presented [1984; 1987], such techniques pretty much fell into oblivion. Instead, visibility was either solved using a depth buffer [Catmull 1974] or using ray tracing [Whitted 1980], and most often with some type of point sampling.

An interesting observation by the computer science community is that the gap between available compute power and memory bandwidth is large, and continues to grow rapidly [Hennessey and Pattersson 2006; Owens 2005]. In addition, the power consumption by a memory access and a floating-point operation differs by at least an order of a magnitude [Dally 2009]. Hence, common advice today is to refactor an algorithm so that it instead uses more computations and fewer memory accesses. With this development of computer architecture, one logical consequence is that it makes more sense to (again) explore analytical visibility computations, which are computationally more expensive than point sampling techniques.

To that end, we present a new visibility engine which is loosely based on previous work on analytical visibility for spatial anti-aliasing [Catmull 1978] and on analytical motion blur with spatial point sampling [Gribel et al. 2010]. Our goal is to generate high-quality images with near-perfect anti-aliasing in both the spatial domain and in the temporal domain. To reduce the dimensionality of the problem, we use line samples [Jones and Perry 2000] in screen space, and solve for analytical visibility along such lines and over time. We present a novel visibility engine for this, and show that our algorithm can rapidly generate practically noise-free images on many-core computers. We also show that a variant of our technique can render ambient occlusion with motion blur.

2 Previous Work

There is a wealth of research in the visibility field, and in this section, we review only the research that is most relevant to our work. This means that we avoid discussing approaches based on point sampling visibility, such as multi-dimensional adaptive sampling [Hachisuka et al. 2008], for example.

Cook et al. [1987] presented the REYES rendering system using rasterization of motion blur and depth of field with stochastic point sampling [Cook et al. 1984]. Lately, much effort has been spent on stochastic rasterization of (micro-)polygons for motion blur and depth of field [Akenine-Möller et al. 2007; Fatahalian et al. 2009; McGuire et al. 2010]. Ragan-Kelley et al. present a hardware architecture for rasterizing motion blur and depth of field [2011]. By decoupling shading from visibility, they essentially extend the concept of multi-sample anti-aliasing to motion blur and depth of field. While REYES' target was offline rendering, this later line of research is targeting interactive or even real-time graphics. In our current work, point sampling is only used for shading.

Catmull [1978] presented a visibility algorithm that processes the polygons from top to bottom, and from left to right in order to exploit coherence of the scene. A per-pixel clipping algorithm is applied where each edge clips the existing polygons into two groups recursively in order to compute the exact polygon coverage, with excellent spatial anti-aliasing as a result. The algorithm by Weiler and Atherton [1977] is rather similar, but also describes how to render shadows and transclucent geometry. Korein and Badler [1983] presented an analytical visibility algorithm for motion blur, where each spatial point sample computed analytical coverage of disks over time. After all geometry had been processed, a hidden surface removal algorithm was applied to resolve for final sample color. The details of how this technique can be extended to linearly moving triangles are given by Gribel et al. [2010]. Korein and Badler were also the first to present accumulation buffering, which can be used for all sorts of point sampling on a per-frame basis. Accumulation buffering has been used for spatial anti-aliasing, depth of field, and motion blur [Haeberli and Akeley 1990; Wexler et al. 2005].

Sung et al. [2002] decouple shading from visibility on a per-object basis, somewhat similar to Burns et al. [2010], and use point sampling for spatial antialiasing, with analytical visibility over time per point sample [Korein and Badler 1983]. In our work, we also use decoupled shading from visibility, and fall back on point sampling of shading, with efficient reuse using a cache [Ragan-Kelley et al. 2011; Burns et al. 2010].

Jones and Perry [2000] presented a "near-analytic" screen-space anti-aliasing technique that uses line samples to reduce dimensionality of the problem. All polygons in the scene are projected onto these lines, and visibility resolved along the line with linear depth per polygon. They use horizontal and vertical line samples through the center of the pixels, and blend the results of these according to the edges inside the pixel. A similar line of work was presented by Gribel et al. [2010], where point sampling was used in screen space, and "line samples" were used to analytically compute motion blur.

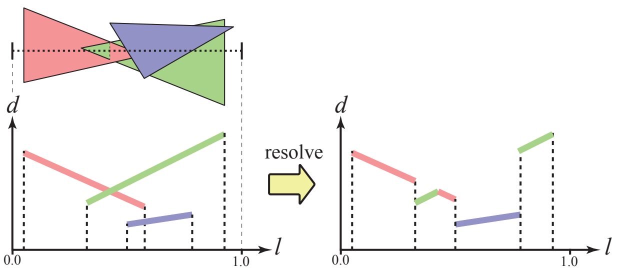

Figure 2: *At the top, a configuration of three triangles (in screen space) intersecting a line sample (dotted line) is shown, and just below, the depth functions, d, for the three triangles on the line sample are illustrated. To the right, we have resolved for the closest depth segment over the parameter, l.*

They also presented a lossy compression algorithm in order to handle a large number of triangles per pixel. In our work, we combine the work of Jones and Perry, Gribel et al., and Catmull to obtain an algorithm for high-quality spatial anti-aliasing with motion blur.

Recently, Manson and Schaefer introduced wavelet rasterization [2011], which can *analytically* rasterize polygons and Bézier curves in two dimensions, and also three-dimensional meshes into voxel grids. They rasterize the primitives into a hierarchical Haar wavelet tree representation and show that it is robust to degenerate input. This is really interesting work that could potentially be applied to motion blur as well. Finally, we refer to the survey by Sutherland et al. [1974] for an overview of ten visibility algorithms pre-dating the depth buffer [Catmull 1974], and to the overview by Sung et al. [2002] for spatio-temporal anti-aliasing.

3 Algorithm Overview

In this section, we present an overview of our algorithm for generating images with high-quality spatio-temporal anti-aliasing. In addition, we will first explain some key concepts around our sampling strategy using analytical visibility computations. We assume that the geometry consists of triangles, and that each triangle vertex can move linearly in world space over time, $t \in [0, 1]$.

Instead of using multiple point samples over a pixel for spatial anti-aliasing, we choose to use a set of line samples [Jones and Perry 2000]. We define a *line sample* as an arbitrary straight line over one or more pixels, and we will compute visibility analytically over such lines. Over each line sample, Jones and Perry determine which triangles overlap with the line sample, and compute the depth, d, at the end points of each visibile segment. The depth, $d = z/w$, is linear in screen space [Blinn 1992], which means that the depth at the end points are sufficient to store. See Figure 2 for an example.

Geometrically, a line sample can be thought of as a three-dimensional triangle with one vertex at the camera position, and going through the end points of the line sample in world space, and extending infinitely. Occasionally, we will refer to the plane of this triangle as the *line sample plane*. A line sample is illustrated as a blue line to the left in Figure 3. We parameterize along each line using a parameter, $l \in [0, 1]$. A core difference compared to other spatio-temporal visibility algorithms is that we resolve visibility in the lt-space, where $t \in [0, 1]$ represents the time dimension, and $l \in [0, 1]$ is the parameter along a line sample. For a static triangle, the intersection between the triangle and the line sample plane will be a single line,[1] which is the same for all values of t. This is illustrated in the lt-space to the left in Figure 3. When a triangle starts to move, however, the moving triangle's line of intersection

[1]Except when the triangle lies in the plane, in which case we will cull it from further processing.

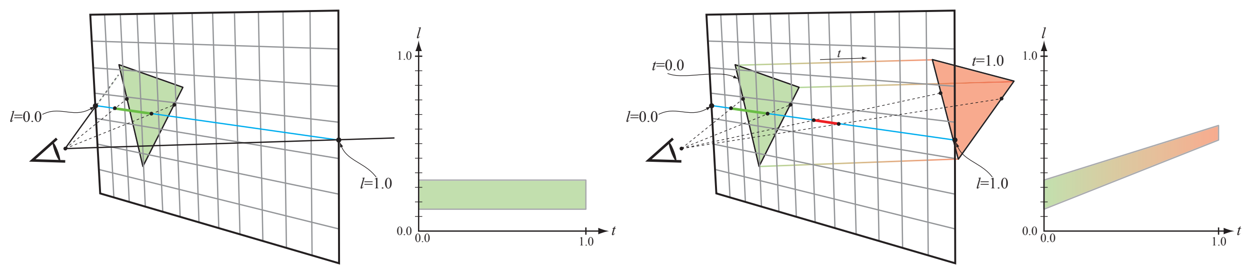

Figure 3: *Left: a static triangle rendered over a 12×7 pixel grid. We show a single line sample (blue line) extending through a row of pixels. The line sample plane is defined by the camera position and the line sample. The lt-space shows the triangle coverage over the line sample and over time, which in this case is a rectangle covering the entire time interval. Right: a triangle being translated over $t \in [0,1]$. The triangle is visualized as green at $t=0$, red at $t=1$, and each vertex moves linearly in between. As can be seen, when the triangle moves, it traces out a different region in the lt-space. Each point, (l,t), will also have a depth, d, but that is not visualized here—see Section 4.*

will change over time, and will trace out a different region in the lt-space. An example is shown to the right in Figure 3.

The motivation for using line samples is manifold. First, a line sample can be thought of as infinitely many point samples on the line, and therefore has potential for fast convergence. Second, as mentioned in the introduction, trading computations for bandwidth is a general advice on today's computing architectures. Finally, using line samples instead of performing a full analytical resolve in the xyt-space reduces the problem from three to two dimensions which makes it more tractable. In practice, we use sampling patterns so that 2–4 line samples will intersect each pixel, and these patterns are discussed in Section 7.

On a high level, our entire algorithm works as follows. First, a bounding volume hierarchy (BHV) is built over all geometry in the scene. A small rectangular region, called a *tile*, of the entire image is rendered at a time. This allows for parallel execution on many threads since a core can render to a tile independently of others. For each tile, the BVH is traversed so that only geometry that overlaps with the tile is processed. In the next step, each line sample inside the tile is processed one at a time. Every triangle that intersects a line sample is represented as a *depth patch* in ltd-space, where each point, (l,t), has a depth, d. These depth patches can form complex surfaces, and in Section 4 and in Appendix A, we present how we construct these patches and approximate them with, what we call, *patch triangles* in the ltd-space. We call our algorithm *semi-analytical*, since most computations are analytical, except for the approximation of the depth patches.

The patch triangles approximating a depth patch are sent to our visibility engine (Section 5), which resolves for depth visibility over the lt-space. This part of our algorithm was inspired by Catmull's analytical screen-space anti-aliasing algorithm [1978]. When all triangles have been processed, final visibility is resolved, and for each pixel, the contribution of the line samples overlapping the pixel is accumulated to that pixel's color. In Section 6, we also show that a variation of our algorithm can be used to compute motion blurred ambient occlusion using "line samples" on the hemi-sphere.

4 Depth Patches

As illustrated in Figure 3, each moving triangle intersecting a line sample will give rise to some region in lt-space. In addition, each point, (l,t), will also have a depth, d. We call these surfaces in ltd-space *depth patches*. Figure 4 shows that these depth patches can form rather complex surfaces, and it is quite obvious that there is little hope in finding an analytical representation of these that also is fast to process. Therefore, our approach is to approximate the exact depth patches with a number of triangles, here called *patch triangles*, in ltd-space. We use the following notation for a triangle in world space. A linearly moving vertex is denoted $\mathbf{p}(t) = (1-t)\mathbf{q} + t\mathbf{r}$, and a triangle is simply three vertices, $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$. Hence, the triangle at $t=0$ is $\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$, and the triangle at $t=1$ is $\mathbf{r}_0\mathbf{r}_1\mathbf{r}_2$. The term *motion vector* is used for $\mathbf{r}_i - \mathbf{q}_i$, and *triangle edge* for any triangle edge at $t=0$ or $t=1$. Next, we describe how our patch triangles are constructed.

Note that the entire surface of a moving triangle consists of the triangles at $t=0$ and $t=1$, and three bilinear patches, each generated by a moving edge. The intersection between such a moving triangle and a line sample plane will trace out one or more connected regions in lt-space. The boundary of such a region is called a *patch outline*, and consists of connected straight lines and curves. This is illustrated in Figure 4. The straight lines originate from the intersections between the line sample plane and the triangles at $t=0$ & $t=1$, and the curved segments will be generated by the intersections between moving edges and the plane. Next, we argue that the points in lt-space, where lines and/or curved segments are connected, are easily generated. We call them *feature points*.

First, we observe that the triangles at $t=0$ and $t=1$ will intersect the line sample plane in lines in the lt-space, and hence the intersections between the plane and the triangle edges are feature points. Second, we note that a moving edge traces out a bilinear patch, which is a ruled surface. This means that the intersection curve(s)[2] between a bilinear patch and a plane will be curves that start and end in the intersection points between the plane and the four edges of the bilinear patch. Hence, these intersection points are also feature points. Therefore, we define the feature points as all the intersections between the line sample plane and the triangle edges & the motion vectors. The triangles at $t=0$ and $t=1$ can *each* give rise to at most two feature points. In addition, each motion vector can give rise to one feature point. In total, this sums to at most $2+2+3=7$ feature points per moving triangle.

These feature points are marked with red crosses to the left in Figure 4, and at each feature point, it is straightforward to compute its depth, d. Depending on the configuration of these feature points, they need to be connected in a certain order. This is an important implementation detail, and so in Appendix A, we describe how the feature points are used to tessellate the true depth patch into a number of *patch triangles*. In that appendix, we also describe how the tessellation can be adapted to rapid change in the depth patch function. For the next section, we assume that our visibility engine receives a stream of patch triangles approximating the true depth patch in ltd-space for each moving triangle.

5 Visibility Engine

The purpose of our visibility engine is to analytically resolve visibility for a spatial line sample and ultimately compute the color

[2]There can be one or two such curves—see Appendix A.

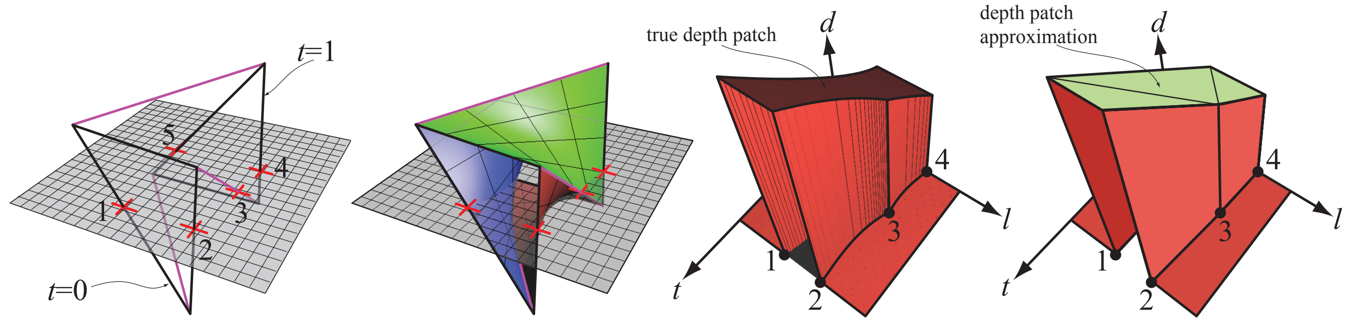

Figure 4: *Left: illustration of a moving triangle intersecting the line sample plane (gray). The triangles at time $t = 0$ and $t = 1$ have black outlines, and the motion vectors are purple. The feature points are marked with red crosses, and are all the intersections between the line sample plane and the triangle edges and motion vectors. Middle-left: visualization of the bilinear patches that are traced out over time with linear motion per vertex. Middle-right: the true depth patch in ltd-space. We generated this image using point sampling. Right: for this depth patch, a possible approximation consists of the three green triangles, called patch triangles.*

contribution from that line sample to each pixel. As mentioned in Section 3, we process a rectangular block of pixels, called a *tile*, at a time. Our geometry is represented in a three-dimensional bounding volume hierarchy (BVH), where each leaf node may contain one or more moving triangles. We cull the BVH against the frustum spanned by the tile, and we further cull the remaining geometry against the line sample plane.

At this point, we have a set of moving triangles that overlap with the line sample, and we would like to find all *visible* patch triangles (Section 4) in both the spatial and in the temporal domain, i.e., in the lt-space. In order to support near-z clipping, all triangles are tested against a line sample plane in view-space. The resulting patch triangles are then individually clipped against near-z, projected into viewport-space, and mapped along the sample line into lt-space. The lt-space is diced up into $m \times n$ uniform grid cells, where m is the number of pixels that the line sample overlaps with, and n is a user-defined constant specifying the number of subdivisions in the t-dimension. The grid cells are referred to as lt-cells, or simply just cells. Our visibility engine operates completely in the lt-space, and can be divided into three stages, namely, *binning*, *depth sorting*, and *pixel integration*. Next, these three stages are described in more detail.

5.1 Binning

The binning stage processes one patch triangle at a time, and finds all patch triangles that overlap with each lt-cell. This is done by conservatively rasterizing the patch triangle to the lt-cells. For all patch triangles overlapping a cell, a pointer to that patch triangle is stored in a list of that cell. In addition, we perform a simple variant of occlusion culling [Greene et al. 1993] in this stage. If a patch triangle is found to completely cover a cell, the maximum depth, $Z^{\text{cell}}_{\max}$, of the patch triangle over the cell is computed. That cell is then listed as "fully covered" with $Z^{\text{cell}}_{\max}$. All subsequent patch triangles that overlap with a fully covered cell can be occlusion culled (and not added to the cell list) if they are farther away than that cell's $Z^{\text{cell}}_{\max}$. In order to occlusion cull even when triangles are not rendered strictly front to back we also store $Z^{\text{cell}}_{\min}$. If a newly added triangle covers the entire cell and its $Z^{\text{tri}}_{\max}$ is less than $Z^{\text{cell}}_{\min}$, the entire cell can be cleared. After all patch triangles have been processed, all potentially visible patch triangles that overlap with a cell in the lt-space are known. It is the task of the following two stages to resolve for final visibility.

5.2 Depth Sorting

The goal of depth sorting is to deliver a list, for each lt-cell, of non-intersecting polygons sorted according to ascending depth. In order to ensure depth order inside each lt-cell, a local BSP-tree [Fuchs et al. 1980] is created from the patch triangles in each cell's list. The patch triangles are added to the BSP-tree in turn, which creates leaf nodes with their splitting plane taken from the patch triangle plane. Subsequent patch triangles may be split against these planes during insertion into the BSP-tree. Once the BSP-tree has been built, depth sorting is simply a matter of traversing the BSP-tree. Note that on average, our BSP-trees are small since they only cover one pixel's extent on a line sample, and a certain span in time. For our test scenes, we usually have less than 100 triangles per lt-cell, and hence creation and traversal is relatively fast.

5.3 Pixel Integration

The goal of the pixel integration is to calculate the color contribution of a sample line to each pixel it overlaps. Once the color for all lt-cells covering a pixel is known, the results can be weighted together to give the final color of the pixel. In order to integrate only the visible part of each polygon, a hidden surface algorithm is used to eliminate occluded geometry in each lt-cell. By traversing the BSP-tree per lt-cell, we obtain a strictly depth sorted list of polygons. This means that once a region in an lt-cell has been found to be covered by a polygon, no other polygon can cover that region.

The hidden surface algorithm is similar to "the pixel integrator" proposed by Catmull [1978]. This algorithm inserts all depth sorted polygons into a tree in order of front to back. Each added polygon is split against the edges of the polygons already in the tree. The part to the left of the edge is added as a child to that edge in the tree. The remaining part is split against the next edge. This is illustrated in Figure 5.

After hidden surface removal, calculating the color of the lt-cell is a simple matter of summing the color of each visible (convex) polygon weighted by its visible area, possibly weighted by a filter kernel. The color of each polygon is determined by invoking the pixel shader. We currently shade each convex polygon at its barycenter, which is a natural place to put it, and this also generated high-quality images. Hence, n is connected to both the number of cells in time as well as the shading frequency (since shading is performed at least once for each patch triangle in each cell) in our current implementation. Texture differentials are calculated analytically using the triangle at $t = 0$.

Discussion As mentioned, our visiblity engine was inspired by Catmull's early work [1978] that operates in the spatial domain (xy) without motion blur. However, for our work, there are a number of subtle but important differences. Our visibility engine and resolve procedure are working entirely in lt-space, which enables motion blur to be handled. In the binning stage, we process one patch trian-

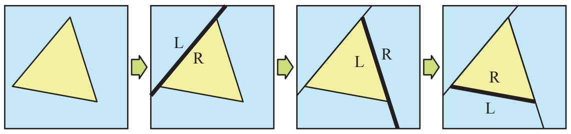

Figure 5: *A patch triangle shown inside an* lt*-cell. In the pixel integrator, the first triangle edge splits the cell into a left (L) region and a right (R) region. In this case, the next edge splits the previous right region into two new regions, and so on. This creates a tree, and each new triangle is clipped against the edges in the tree, and possibly inserted into the tree.*

gle at a time until all patch triangles have been completely binned, which makes our engine similar to feed-forward rasterization. This is in contrast to Catmulls' scanline order processing which requires sorting on l, and handling of every patch triangle covering a scanline before proceeding to the next. In addition, we have added a simple form of occlusion culling for better efficiency. All images in Catmull's paper [1978] appear to consist of 2D layers composited with anti-aliasing, but we need general sorting, and therefore use a BSP-tree instead of assuming already sorted geometry.

6 Ambient Occlusion

In this section, we will show that, with some small modifications, our depth patches (Section 4) and our visibility engine (Section 5) also can be used to compute *motion blurred* ambient occlusion (AO). To the best of our knowlege, this is a topic that has received very little attention in graphics research. We start with a brief introduction to ambient occlusion, and then describe how our algorithms can be used for static and motion blurred ambient occlusion in subsequent subsections. Zhukov et al. [1998] define the amount of ambient occlusion, o, as:

$$o(\mathbf{p}, \mathbf{n}) = \frac{1}{\pi} \int_{\Omega} \rho\left(d(\mathbf{p}, \omega)\right)(\omega \cdot \mathbf{n}) d\omega, \qquad (1)$$

where $\mathbf{p}$ is the point where AO is computed and $\mathbf{n}$ is that point's normal. The integral is over the hemisphere, Ω, with its "north pole" in the direction of $\mathbf{n}$. The distance function, d, returns the distance to the closest occluder in direction ω, and ρ is a distance fall-off function, which has a value between 0 (fully occluded) and 1 (not occluded). Intuitively, the equation computes how much of a white hemisphere that a diffuse receiver can "see." For high-quality renderings, one may need a thousand rays to compute AO using point sampling [Laine and Karras 2010], and for motion blurred point-sampled AO, we expect that even more rays may be needed.

6.1 Static Ambient Occlusion

Instead of integrating over the hemisphere using point samples, we extend the concept of line samples to the hemisphere. Here, we define a line sample as the intersection between the hemisphere, and a plane going through the center of the hemisphere. An example is the orange plane shown to the left in Figure 6. In the following, we will first concentrate on a single line sample. However, note that a number of line samples will be needed to accurately sample AO over the hemisphere, and we will return to line sample distributions later in this section. For all triangles above or overlapping the hemisphere base plane (going through the "equator"), we compute the intersection (if any) between the triangle and the line sample plane, and project the intersection onto the line sample's hemicircle. The next step is to project that down to the base plane as shown to the right in Figure 6, and there we illustrate our parameterization, l, of the line sample on the hemisphere. Note that this implements the Nusselt analog, which means that the cosine factor of Equation 1 is included by design per line sample.

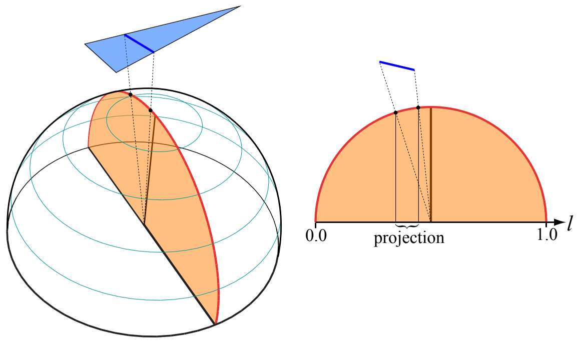

Figure 6: *Our approach for ambient occlusion is to use a number of line samples over the hemisphere. In this illustration, only a single line sample is shown, but in practice, several will be used to cover the hemisphere. Left: a line sample in this case corresponds to a hemi-circle (red), and all triangles overlapping with the plane (orange) through that hemicircle are projected onto the hemicircle. Right: illustration of the projection of an intersection between a triangle and the line sample plane in two dimensions. As can be seen, the intersection is first projected onto the hemicircle, and then down to the black line, which is parameterized by* l*.*

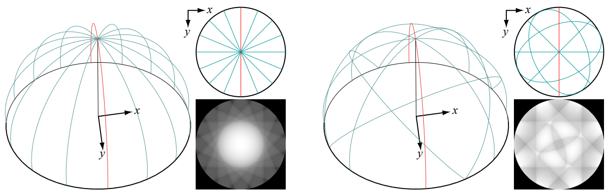

Figure 7: *Left: a line distribution with eight rotated planes going through the north pole of the hemisphere. The lines projected to* xy*, and their line density image [Grabli et al. 2004] are also shown. Since this distribution is rather non-uniform, we remap the coordinates along* l *in order to obtain a uniform distribution. Right: a more uniform distribution of lines in the* xy *circle of the hemisphere. As can be seen, the line density image is more uniform for this distribution.*

Similar to line sampling in screen space, we also use linear depth segments and resolve for visibility with respect to depth exactly as shown in Figure 2, and described by Gribel et al. [2010]. However, note that depth is not linear over l in this case. If the goal is to only compute whether the hemisphere is occluded or not, then it does not matter that depth is not linear, since we are only interested in occlusion and not depth. However, if the falloff function, ρ, in Equation 1, needs to be taken into account, depth matters. In this case, long projections on l must be diced up into several shorter linear depth segments, which makes for a better approximation.

The remaining part at this point is line sample distributions over the hemisphere. As far as we know, this is a rather unexplored topic in sampling. To the left in Figure 7, a simple sampling pattern is shown, where all eight line samples are passing through the "north pole" of the hemisphere. In general, we strive after uniform distributions in xy-space, and it is quite clear that this distribution is non-uniform. To counteract that, we instead redistribute the projection on l before they are inserted. In this case, this amounts to a quadratic remapping function, similar to that used for depth of field remapping [Akenine-Möller et al. 2007]. Without remapping, we also measure line density as suggested by Grabli et al. [2004], where we used a measurement radius of 0.5. Such images are also shown in Figure 7. As can be seen to the right in the same figure, a more uniform distribution can be obtained if the line sample planes also are allowed to tilt. This makes the mapping parameterization a bit more complex, but we expect that it will be worth the effort.

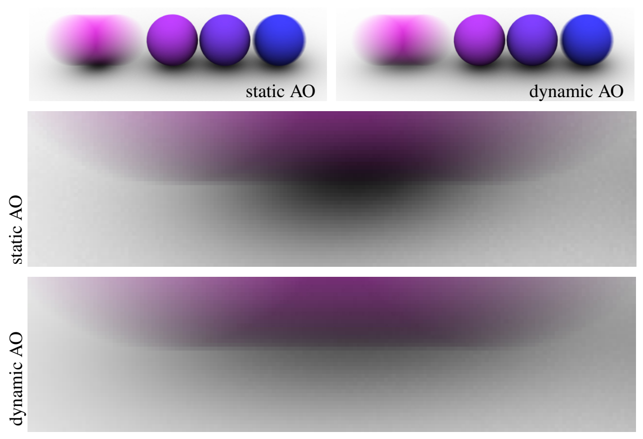

Figure 8: *Static versus motion blurred (dynamic) ambient occlusion (AO) with 16 line samples for AO. For the static case, AO was computed at $t = 0.5$. Dynamic AO renders more plausible images as seen in the close-ups.*

6.2 Motion Blurred Ambient Occlusion

Our extension to motion blurred ambient occlusion (AO) is rather straightforward. Instead of just processing visibility along the l-axis (Section 6.1), we now compute visibility in lt-space as described in Section 4 and 5. For static receivers of AO, this technique works as expected. A moving sphere over a plane will cast a motion blurred AO shadow on the static plane receiver, for example.

For dynamic receivers, this situation is more complex, because both the receiving sample position on a surface, and its normal may change as a function of time. We avoid this complexity by considering all motion relative to a local coordinate system at the sample point. The occluders are transformed into this local coordinate system at both $t = 0$ and $t = 1$. The relative motion of each vertex is approximated as linear with respect to time, which we believe is reasonable because that is exactly how vertex motion is described in our system. Note also that we clip the patch triangles against $z = 0$ so that parts of patch triangles "below" the hemisphere are not processed.

As described in Appendix B, we use a shading cache in order to efficiently reuse shaded values over the time dimension. We note that shading caches [Ragan-Kelley et al. 2011; Burns et al. 2010] are two-dimensional over some parameterization of the rendered surfaces, and for motion blur, the time, t, is collapsed and always computed at $t = 0$, for example. This has direct consequences for shadows and AO when they are point sampled in time. Shadowing at a moving surface point will simply be evaluated at $t = 0$, and if the surface point later in time changes from being in shadow to being lit, that will not be accounted for. This can easily give rise to images with unnatural looking shadows. This is a limitation of shading caches, and it may be an interesting avenue for future work to extend shading caches to handle motion blurred shadows and AO better.

Interestingly, our solution will not compute AO at $t = 0$ and reuse this value over all samples. As described above, our algorithm will compute the AO in lt-space, and it is the average AO over time that will be computed and put into the shading cache. For static receivers, this generates a correct image. However, for moving receivers, this is not strictly correct, but we believe the average over time is a better solution than computing AO at a single instant of time and reuse. In Figure 8, we show the difference between static AO and motion blurred AO.

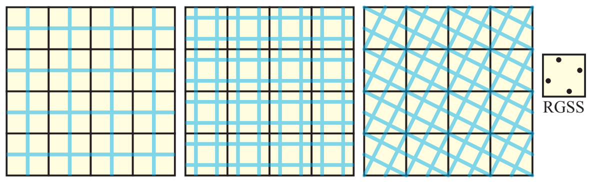

Figure 9: *Line sample patterns for 4×4 pixels. Left: one horizontal and one vertical (1H1V) line sample (blue) is going through each pixel. Middle: two horizontal and two vertical (2H2V) line samples. Right: a rotated grid of line samples (RGLS), inspired by the point sampling pattern called RGSS.*

7 Implementation

We have implemented all our algorithms in a custom renderer in C++, and have the pixel processing part of the algorithm threaded in order to exploit more than one CPU core. Our algorithm is compared against stochastic rasterization with efficient backface and view frustum culling. In addition, we have an early-Z depth test and a simple form of occlusion culling [Greene et al. 1993] with one $Z_{\max}$-value per 8×8 pixels. For stochastic rasterization, we use multi-jittered sampling points. In the following, we will discuss line sampling patterns for screen space sampling, which is not a well explored topic in graphics.

Some obvious line sampling patterns to test include using one horizontal and one vertical line sample per pixel. This is what Jones and Perry used in their inspirational research [2000]. Instead of creating two unique lines per pixel, we extend the lines so they start and end at the sides of the current tile in order to better exploit coherency. In general, this reduces the number of BVH traversals and patch triangle generations needed per rendered image. For better quality, one can increase to two horizontal and two vertical line samples per pixel. Such schemes are shown to the left in Figure 9. However, it is well-known that humans are most sensitive to jaggies on near-horizontal and near-vertical edges, and thereafter on edges which are close to $45°$ [Naiman 1998]. This clearly motivates low-cost point sampling patterns, such as rotated grid super sampling (RGSS). To that end, we experimented with a line sampling pattern that exhibit similar charateristics as RGSS, but is also designed to share long line samples across many pixels. This pattern, which we call rotated grid line sampling (RGLS), is shown to the right in Figure 9. While RGLS perform much better compared to using horizontal and vertical line samples, we leave the optimization of line sample patterns for future work. This is an important topic that we intend to revisit in later research.

8 Results

We have rendered our images using a Mac Pro 5.1 with two six-core Intel Xeon CPUs at 2.93 Ghz and 8 GB of memory. We use a decoupled shading cache as described in Appendix B for both our algorithm and for stochastic rasterization of motion blur. For one core, we shade about 10% of the requested shading values for 64 stochastic samples. This is deliberately rather high because we want high-quality shading. For 24 threads, this increases to 21% due to the fact that some shading computations are computed more than once (see Appendix B). Some details of our stochastic rasterization implementation are given in Section 7. In all our renderings, we use a tile size of 32×32 pixels. Concerning tile sizes, we have noted that our algorithm is not very sensitive to bin spread. For example, rendering the chess scene at 512×512 resolution using 16×16 tiles requires only 30-35% more time than using 512×512 tiles, which is equivalent to sort-last. We have used three main test scenes to evaluate our algorithm. The *chess* scene in Figure 1 has 29,068 triangles and procedural texturing, *Sponza* in Figure 11 has

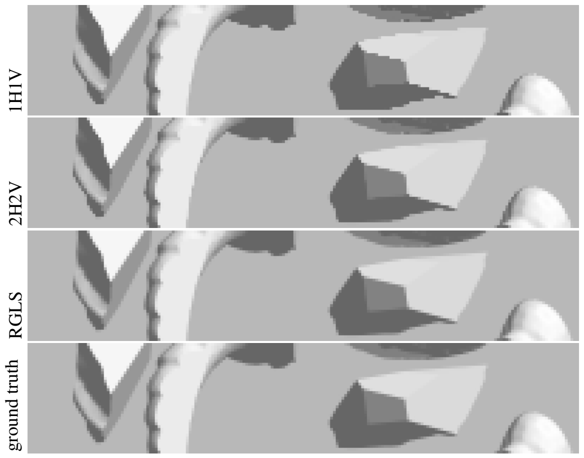

Figure 10: *Here we compare the line sampling pattern quality for a static frame of the breaking armadillo. As can be seen, rotated grid line sampling (RGLS) is substantially closer to the ground truth (256 stochastic samples) than the other two.*

66,450 triangles with lighting pre-baked into textures, and *breaking armadillo* in Figure 12 has 88,120 triangles (see video). The majority of the triangles in chess and armadillo are subpixel-sized at 1024×768 pixels. In addition, we constructed two special scenes for a detailed performance analysis; the *trees* scene with 273,468 triangles and high depth complexity (see Figure 13), as well as a simple *plane* scene with varying tessellation (see Figure 14).

The line sample patterns for spatial anti-aliasing in Figure 9 are evaluated in Figure 10 for one static frame in the breaking armadillo scene. As can be seen, the RGLS scheme is superior to the other two. The main reason for this is that rotated patterns are better at combating aliasing on near-horizontal and near-vertical edges, which visually suffer the most from aliasing [Naiman 1998]. In general, a pattern's edge anti-aliasing effectiveness depends on the angle between the lines of the sampling pattern and the edges of the geometry to be rendered. Furthermore, in our case, there is a small rotated square in the middle of each pixel, and a triangle that falls completely within this area can disappear. However, this is usually not a problem for connected geometrical objects, but can generate artifacts for single, disconnected triangles. In terms of performance, one horizontal and one vertical (1H1V) line sample per pixel is about twice as fast as two horizontal and two vertical (2H2V), which is to be expected since there are twice the number of line samples in 2H2V. In addition, 2H2V is about 10% faster than RGLS due to the fact there are more and longer (on average) line samples per tile in RGLS compared to 2H2V.

Since motion blurred visibility is the major focus of our paper, we first report rendering times without ambient occlusion for our three test scenes, all at 1024×768 pixels with RGLS. As seen in Figure 1, we can render the chess scene in 3.6 seconds (s) with our algorithm. The number of subdivisions in time (see Section 5) was set to $n = 1$, which worked well due to rather low-frequency shading. An image with 49 samples per pixel using stochastic rasterization (SR) took 3.8s to render, and for 256 samples, 27s were needed. If we increase the amount of motion by 50%, our algorithm renders the chess scene in 4.6s, while SR uses 4.0s. However, the SR generated image contains substantially more noise (see video) in the regions with motion, while our image remains free of noise. Hence, if noise-free images are needed, it is clear that our visibility algorithm is very competitive.

The Sponza scene, in Figure 11, has camera motion. For our algorithm, we have implemented near-plane clipping, but we have not done so for stochastic rasterization (SR). Therefore, we cannot render Sponza with SR, and we note that this actually gives SR a little speed advantage in our other timings, since clipping is not done for SR. Here, we used $n = 16$ to ensure high-quality renderings. At 1024×768, this image took 20.1s to render. In Figure 12, we show the breaking armadillo scene. As can be seen, the break-even point compared to stochastic rasterization is around 144 samples per pixel, which is much higher than for the chess scene. This is due to the armadillo having more triangles that also are located in a small volume of the scene.

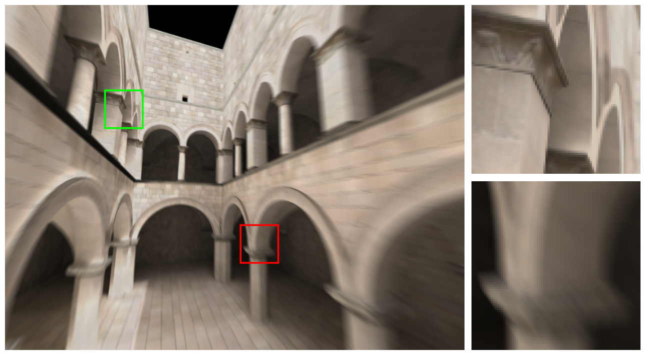

Figure 11: *The Sponza scene with camera motion, rendered with our algorithm, and with close-ups on both geometric edges (top right) and on a region with lots of motion (bottom right).*

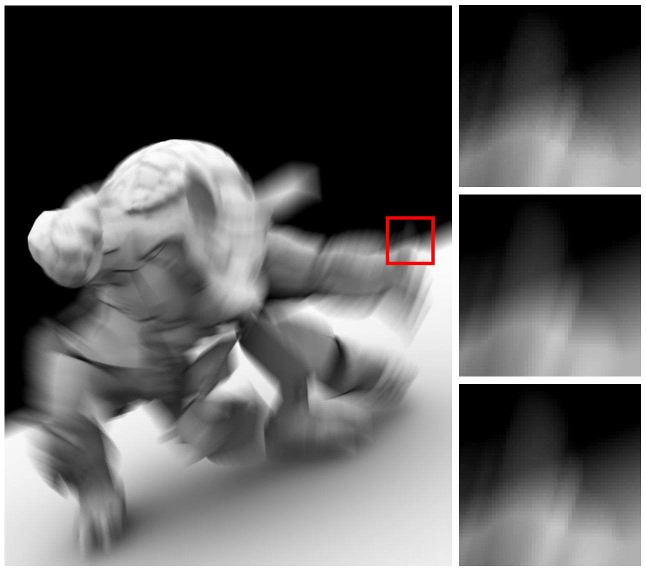

Figure 12: *Left: Armadillo with ambient occlusion with 256 stochastic samples. The images to the right are close-ups of armadillo's moving arm without ambient occlusion. Right-top: using 144 samples per pixel with stochastic rasterization rendered in 10.4s. Right-middle: close-up rendered with our analytical algorithm in 9.8s. Right-bottom: ground truth with 625 samples.*

In order to investigate how our algorithm behaves for high depth complexity in combination with increasing motion, we measured rendering time with respect to increasing motion for various values of n. This is shown in Figure 13. The results suggest that performance scales linearly with increasing motion, given that an optimal value of n is chosen. This implies that an adaptive splitting scheme in time is worth investigating in future work.

In Figure 14, a more detailed performance analysis of the different stages of the algorithm is shown. The plane scene is rendered using a single core with different tessellation rates as well as different values for n. Depth sorting and hidden surface removal are clearly

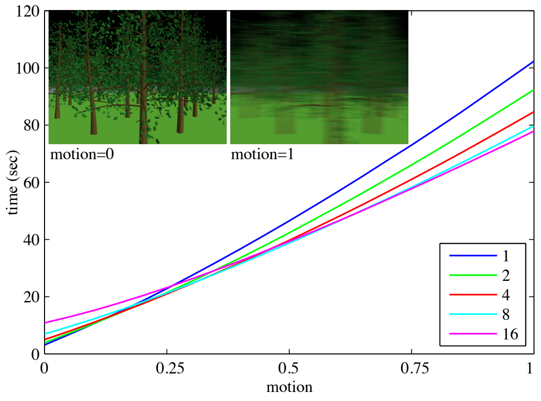

Figure 13: *Measurements from the trees scene, rendered with increasing panning motion. The different lines represent n = 1, 2, 4, 8 and 16.*

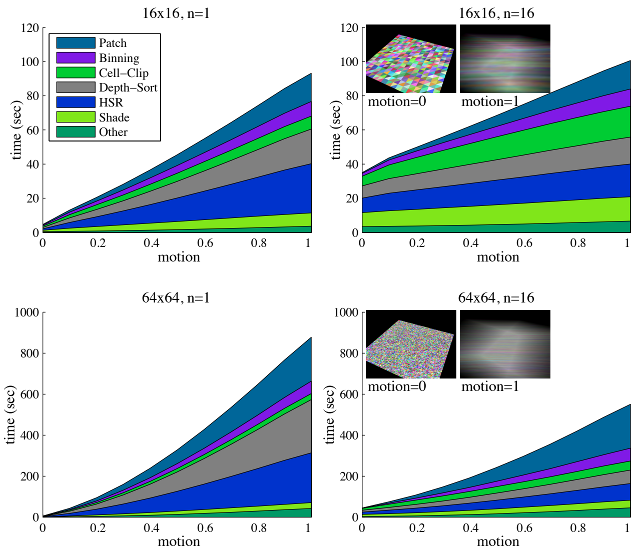

Figure 14: *Detailed measurements of single-core rendering of the plane scene with increasing panning motion. The different graphs represent various combinations of tessellation rate (16×16 and 64×64) and values for n (1 and 16). The "cell-clip" area corresponds to clipping of binned triangles to the cell boundaries, before depth sorting. The "other" area consists of vertex shading, BVH construction, BVH traversal, tile setup, line setup and tile to frame buffer transfer.*

the bottlenecks with high tessellation and motion when using a low value of n. These stages are most affected by a high number of triangles per lt-cell. As expected, increasing n greatly reduces the aforementioned stages. Patch generation time appears to be super-linear with increasing motion. The reason for this is that our BVH becomes less efficient for large amount of motion.

To analyze the approximation error introduced by our algorithm, we made a comparison to a stochastic rendering of the chess scene. The differences in depth, d, were computed and expressed as a fraction of $Z_{\text{far}} - Z_{\text{near}}$, and compiled into a histogram as shown in Figure 15. Note that our approximation also introduces errors in lt, i.e., the patch outline, and this may alter the geometrical silhouette, which means that sometimes the correct geometrical object will be missed. In such cases, the depth errors may be substantial. Nevertheless, for zero subdivisions of the patch outline (described in Appendix A.1), 99.82% of the errors fall within $< 0.1\%$ of $Z_{\text{far}} - Z_{\text{near}}$. For three subdivisions, this increases to 99.95%. This implies that the approximations are reasonable. The heat maps

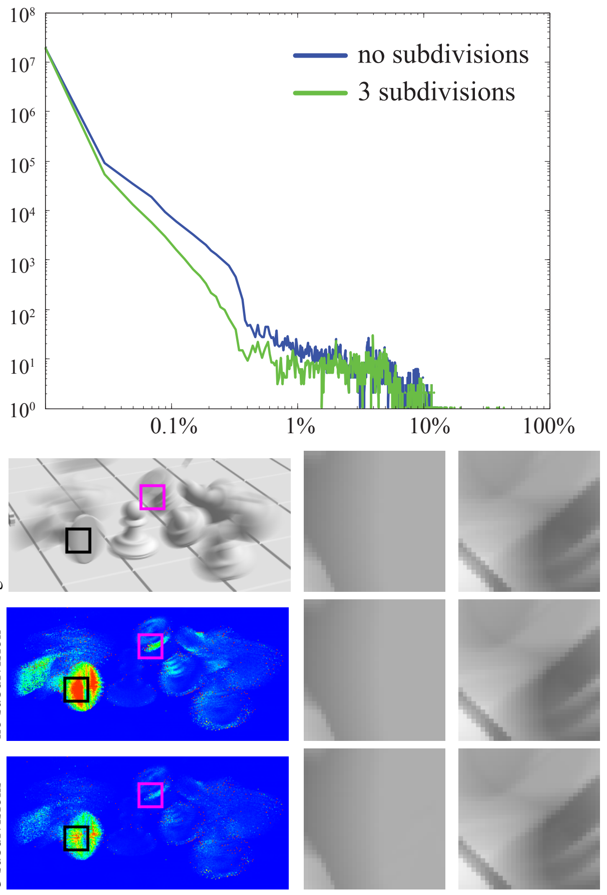

Figure 15: *Measurements and visualizations of introduced depth-approximation errors compared to a stochastic rendering. The loglog-histogram displays depth error as a fraction of the near-to far-plane distance. As is expected, the plots have identical integrals, even though this fact is somewhat obscured in the figure due to the logarithmic scale. Heat maps and zoom-ins of the scene then further detail distribution and magnitude (in log-scale) of the errors, as well as visual comparisons to the ground truth image.*

of the scene in Figure 15 visually shows the distribution and magnitude of the errors. More importantly, a direct comparison between our rendered images to the ground truth is included. One area with high-density errors is the pedestal bottom of the tumbling chess piece to the left. Here, depth varies significantly over time, which causes greater errors due to linearisation, but as can be seen, visual errors are still hard to detect.

With some adjustments, our visibility algorithm can also be used to compute ambient occlusion (AO) both for static and for motion blurred scenes. This is a nice side effect described in Section 6. Except for point sampling AO in time, we are not aware of any algorithms that specialize in motion blurred AO, and yet, the difference in the rendered images can be substantial, as shown in Figure 8 and 16. In all our renderings with AO, we used line sampling distributions where all line samples go through the "north pole" of the hemisphere, as shown to the left in Figure 7. As expected, doubling the number of line samples doubles the time spent on computing AO. Renderings with different number of line samples are shown in Figure 17. For the chess scene with eight line samples for AO and $n = 4$, rendering with static AO took 251 seconds, and with motion blurred AO, this increased to 2075 seconds. We have seen similar increase in rendering times for the other scenes. The focus of our work has been on motion blurred visibility in screen

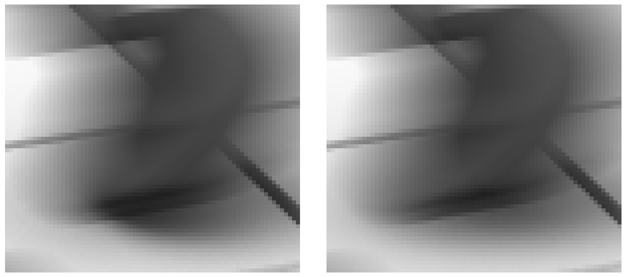

Figure 16: *A close-up of the topmost pawn in Figure 1, with static ambient occlusion (left) and motion blurred ambient occlusion (right). Both images were rendered with 16 line samples for ambient occlusion. Note that in cases like this, the difference is rather large.*

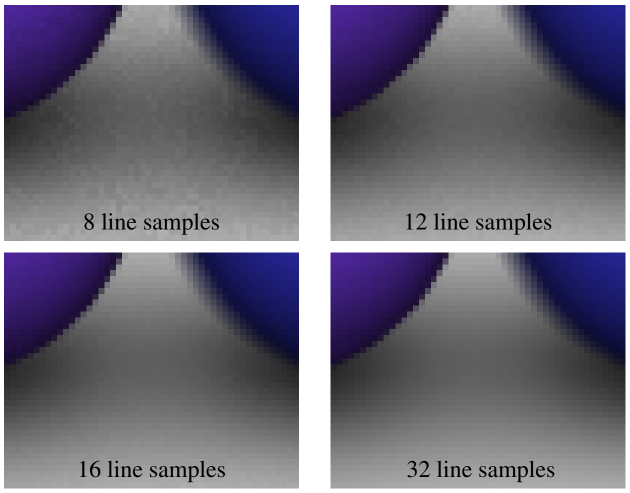

Figure 17: *In these images, we illustrate the effect of increasing the number on lines samples of the hemisphere for ambient occlusion.*

space with spatial anti-aliasing, and hence, the evaluation of AO is not as rigorous. In future work, we will therefore compare both rendering speed and image quality of our AO algorithms (both static and motion blurred) against stochastic point sampling, and test line sampling patterns where the planes are allowed to tilt (right in Figure 7), and optimize the placement of the planes. In addition, we will also optimize for rendering speed.

9 Conclusions and Future Work

To the best of our knowledge, we have presented the *first* visibility algorithm that computes semi-analytical motion blur over spatial line samples, and we have shown that essentially noise-free images can be generated with competitive rendering times. By using line samples in the spatial domain, our visibility problem for motion blur becomes a two-dimensional problem, which makes it tractable and efficient. The approximation we have introduced is in the depth function of a moving triangle over a line sample, and despite this, we have shown that our approximation generates high-quality images. This approximation is the reason why we choose to call our algorithm *semi-analytical*. Our algorithm can also be used for rendering of motion blurred ambient occlusion with high quality. In contrast to our work, the previous methods we are aware of for this are based on point sampling, and may require 512–1024 point samples for high quality for *static* scenes alone [Pantaleoni et al. 2010].

We believe our research opens up a wide range of interesting future work to be done. If point sampling is used in the spatial domain, we believe that our algorithm can be immediately used for computing semi-analytical (SA) depth of field, which is a two-dimensional problem. With our current algorithm, we can also approximate visibility over a lens with a set of line samples, which is similar to previous work [Akenine-Möller et al. 2007], and also use line samples in the spatial domain. Extending with another dimension may be possible, and would open up for even more usage areas, such as SA depth of field with spatial line samples, or pixel area integration with SA motion blur, or SA depth of field with motion blur with spatial point sampling. While sampling patterns always are important in order to generate high-quality images, our focus has not been on developing high-quality line sampling patterns. Instead, our main focus has been on the creation of depth patches and the visibility engine. However, this is an interesting and important topic, and we will revisit this in future work. We would also like to integrate the occlusion culling more with the pixel integrator so that culling will work more efficiently for micropolygons.

Acknowledgements We acknowledge support from the Swedish Foundation for Strategic Research, the Swedish Research Council, and Tomas is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation. The Armadillo comes from the Stanford University Computer Graphics Laboratory.

References

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, 7–16.

BLINN, J. 1992. Hyperbolic Interpolation. *IEEE Computer Graphics and Applications, 12*, 4 (July), 89–94.

BURNS, C. A., FATAHALIAN, K., AND MARK, W. R. 2010. A Lazy Object-Space Shading Architecture With Decoupled Sampling. In *High Performance Graphics*, 19–28.

CATMULL, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

CATMULL, E. 1978. A Hidden-Surface Algorithm with Anti-Aliasing. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, 6–11.

CATMULL, E. 1984. An Analytic Visible Surface Algorithm for Independent Pixel Processing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, 109–115.

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

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, 96–102.

DALLY, W. 2009. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation).

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *High Performance Graphics*, 59–68.

FUCHS, H., KEDEM, Z., AND NAYLOR, B. 1980. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (Proceedings of SIGGRAPH 80)*, vol. 14, 124–133.

GRABLI, S., DURAND, F., AND SILLION, F. 2004. Density Measure for Line-Drawing Simplification. In *Pacific Graphics*, 309–318.

GRANT, C. W. 1985. Integrated Analytic Spatial and Temporal Anti-Aliasing for Polyhedra in 4-Space. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, 79–84.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH*, 231–238.

GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. 2010. Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics*, 163–172.

HACHISUKA, T., JAROSZ, W., WEISTROFFER, R., K. DALE, G. H., ZWICKER, M., AND JENSEN, H. 2008. Multidimensional Adaptive Sampling and Reconstruction for Ray Tracing. *ACM Transactions on Graphics, 27*, 3, 33.1–33.10.

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

HENNESSEY, J. L., AND PATTERSSON, D. A. 2006. *Computer Architecture: A Quantitative Approach*, 4th ed. MKP Inc.

JONES, T. R., AND PERRY, R. N. 2000. Antialiasing with Line Samples. In *Eurographics Workshop on Rendering*, 197–205.

KOREIN, J., AND BADLER, N. 1983. Temporal Anti-Aliasing in Computer Generated Animation. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, 377–388.

LAINE, S., AND KARRAS, T. 2010. Two Methods for Fast Ray-Cast Ambient Occlusion. *Computer Graphics Forum (Eurographics Symposium on Rendering), 29*, 4, 1325–1333.

MANSON, J., AND SCHAEFER, S. 2011. Wavelet Rasterization. *Computer Graphics Forum, 30*, 2, 395–404.

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Hardware-Accelerated Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics*, 173–182.

NAIMAN, A. C. 1998. Jagged Edges: when is Filtering Needed? *ACM Transactions on Graphics, 17*, 4, 238–258.

OWENS, J. D. 2005. Streaming Architectures and Technology Trends. In *GPU Gems 2*. Addison-Wesley, 457–470.

PANTALEONI, J., FASCIONE, L., HALL, M., AND AILA, T. 2010. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM Transaction on Graphics, 29*, 3, 37.1–37.10.

RAGAN-KELLEY, J., LEHTINEN, J., CHEN, J., DOGGETT, M., AND DURAND, F. 2011. Decoupled Sampling for Graphics Pipelines. *to appear in ACM Transactions on Graphics, 30*, 3.

SUNG, K., PEARCE, A., AND WANG, C. 2002. Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics, 8*, 2, 144–153.

SUTHERLAND, I. E., SPROULL, R. F., AND SCHUMACKER, R. A. 1974. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys, 6*, 1, 1–55.

WEILER, K., AND ATHERTON, P. 1977. Hidden Surface Removal using Polygon Area Sorting. In *Computer Graphics (Proceedings of SIGGRAPH 77)*, 214–222.

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

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. *Communications of the ACM, 23*, 6, 343–349.

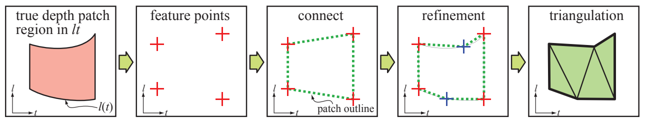

Figure 18: *Overview of depth patch triangulation.*

ZHUKOV, S., IONES, A., AND KRONIN, G. 1998. An Ambient Light Illumination Model. In *Eurographics Workshop on Rendering*, 45–55.

A Depth Patch Approximation

In this appendix, we will describe how our patch triangles, which approximate a depth patch, are generated. We start by deriving a formula for $l(t)$, which describes when a moving triangle edge intersects with the line sample plane. This function is needed to adaptively refine the approximation of the depth patch.

Recall that a vertex is described by $\mathbf{p}(t) = (1-t)\mathbf{q}+t\mathbf{r}$, $t \in [0,1]$. A moving edge, defined by $\mathbf{p}_0$ and $\mathbf{p}_1$, is then a bilinear patch: $\mathbf{b}(s,t) = (1-s)\mathbf{p}_0(t) + s\mathbf{p}_1(t)$. Without loss of generality, we assume that the line sample plane is $y = 0$, and that the l-parameter coincides with x. By setting the y-component of $\mathbf{b}(s,t)$ to 0.0, we can obtain an expression for s:

$$s = -\frac{p_{0y}}{p_{1y} - p_{0y}}, \tag{2}$$

where $p_{0y} = (1-t)q_{0x} + tr_{0x}$, etc. Replacing s by the equation above in the expression for the x-component of $\mathbf{b}(s,t)$, we obtain $l(t)$, after some simplification, as:

$$l(t) = \frac{p_{0x}p_{1y} - p_{0y}p_{1x}}{p_{1y} - p_{0y}}, \tag{3}$$

which clearly is a rational polynomial in t, where the numerator is of degree two, and the denominator is of degree one.

An overview of our depth patch triangulation algorithm is shown in Figure 18. Briefly, we first compute the *feature points* as described in Section 4. These are then connected as described below, and if needed, additional points are added adaptively in a refinement step. Finally, triangulation produces the final patch triangles.

Given a number of feature points in lt, each with a certain depth, d, the goal is now to form a triangulated approximation using these three-dimensional points. This is carried out by considering one moving edge at a time, and based on its interaction with the line sample plane as it moves over time, making connections between the feature points. The final set of connections outline one or many disjoint depth patches. In the final step, these patch outlines, shown in the middle in Figure 18, are triangulated.

In the following, we describe how the feature points are *connected*. If the triangle at $t = 0$ or $t = 1$ intersects the line sample plane, one or two feature points are generated. In the case of two feature points, it is obvious that they should be connected and form an edge of the patch outline. However, we also observe that each moving edge may intersect the line sample plane, and trace out a curve in lt, and its corresponding feature points should also be connected to form part of the patch outline. If a moving edge generates two feature points, they must simply be connected. See Figure 19. When there are four feature points, the situation is a bit more complex, and the solution is illustrated to the right in the same figure.

Four feature points is an indication that the edge initially intersects the plane, turns parallel to it and then intersects the plane again in the opposite direction. At the point, t_s, of edge-plane parallelism, there will be no (edge is above or below plane) or infinitely many

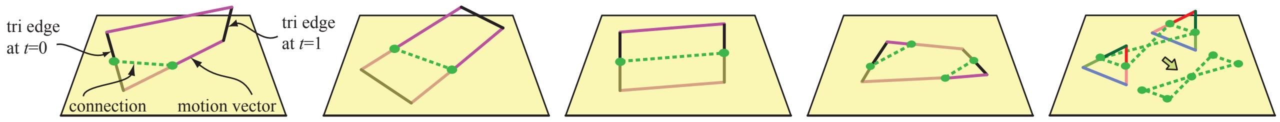

Figure 19: *Connections schemes for moving triangle edges. When there are two feature points (green circles), the feature points are simply connected. We show three possible situations that can occur (first three images). In the fourth image, the moving edge has generated four feature points. In this case, the two earliest (in t) feature points are connected, and then the remaining two. This assures that no connection will "bridge" the singularity point where the edge is parallel to the plane. The final image shows a situation where the facing of the triangle changes as it moves. Here, the connections will overlap in* lt*-space, and to resolve this, the patch is split by adding the intersection point.*

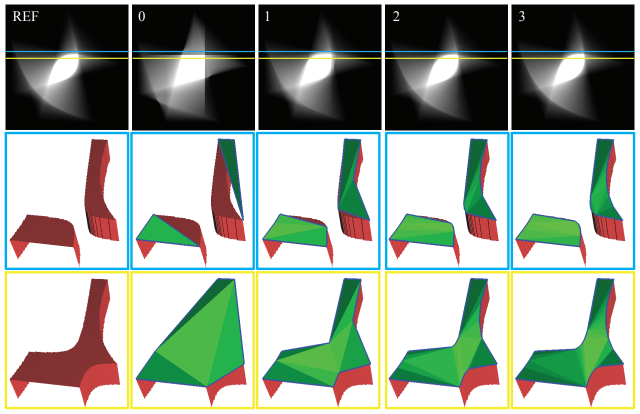

Figure 20: *At the top left, we show a rotating white triangle rendered with 625 stochastic samples, i.e., a reference image. Below that image, we also show the corresponding depth patches (generated using dense point sampling) for the yellow and blue lines in the image. For the top line sample, note how the line will reside completely outside the triangle, during a short interval, which generates two disjoint depth patches (shown in the middle row).*

(edge is in the plane) intersection points, thus making $l(t)$ undefined for $t = t_s$. Two feature points will reside on each side of t_s, that is, at $\leq t_s$ (initial intersection) and $\geq t_s$ (second intersection), respectively. By connecting the feature points on each side of t_s, we avoid letting any connection span undefined values of t. In the extreme, when the edge is precisely in the plane at t_s, two feature points will occur at t_s, and will thus constitute end point for one of the connections and start point for the other. Hence, each connection will span an interval (t_a, t_b) (not including the end points) that is defined in $l(t)$.

After all connections have been made, we have a set of lines, and we simply create the patch outline (middle in Figure 18) by connecting lines that share feature point vertices. In a final step, the patch triangles are created by triangulating the resulting polygon(s). Our approach to generate the patch triangles works really well for triangles that undergo moderate rotation, however, the approximation may be too crude for extreme motion. To that end, we present an adaptive refinement procedure.

A.1 Adaptive Refinement of the Approximation

Better approximation accuracy can be achieved by adding more points to the patch outline before triangulating. This is done by exploiting that $l(t)$ (Equation 3) is defined, and monotonic in $\frac{\partial l}{\partial t}(t)$, within each connection range (t_a, t_b). Hence, additional points can be created by evaluating $l(t)$ at arbitrary values of $t \in (t_a, t_b)$. A good choice of t is one that captures as much of the curvature of $l(t)$ within the range as possible. In our approach, we interpolate the angle of the tangent slopes in the end-points of the range, then solve $\frac{\partial l}{\partial t}(t)$ for this value to retrieve t, which we use to generate a new point on the patch outline. If just one extra point is to be obtained, the average of the two angles is used, and the theory goes as follows. Given an interval (t_a, t_b) for a connection, the angles, α_a & α_b, of the slopes at t_a and t_b are:

$$\alpha_i = \arctan\left(\frac{\partial l}{\partial t}(t_i)\right), \text{ for } i = \{a, b\}. \tag{4}$$

Using the arithmetic mean of the angles, we then need to solve the following equation for t:

$$\frac{\partial l}{\partial t}(t) = \tan\left(\frac{\alpha_a + \alpha_b}{2}\right) \tag{5}$$

Since $\frac{\partial l}{\partial t}(t)$ is quadratic in t, this will result in up to two solutions, but only one can be part of (t_a, t_b). By evaluating $l(t)$ for this new t, a new approximation point is acquired. For additional points, further subdivision can be made adaptively. Examples of termination criterion are maximum number of new points, depth error tolerance etc. As the number of subdivisions grows toward infinity, the approximation outline approaches the analytical solution in lt-space. In Figure 20, we show a triangulated depth patch for different levels of subdivision. Note, however, that the depths will still be linearised over the interior of this outline.

B Shading Cache

We use an object-space shading cache [Burns et al. 2010] to avoid excessive shading, and we extend their cache with an image pyramid for efficient memory usage. Note that we use the same cache for all our rendering threads, and hence need to allocate 2.0 GB for the entire cache for our renderings. Each object's shading values are stored in a map, which is represented by 16×16 independent submaps, where each submap is responsible for a region of the uv-space. Each submap consists of a pyramid of cached shading values, which is allocated lazily. At the top of the pyramid, a single shading sample represents the entire uv span of the submap. This is the lowest shading resolution available. Each lower level in the pyramid has double the resolution of the previous level.

We use the derivatives of the barycentric coordinates, u and v, with respect to screen space, x and y, when selecting appropriate pyramid levels. This is analogous to standard mipmapping. When a shading request is processed, the correct pyramid level is selected based on the these derivatives, and then the uv-coordinates of the shading request is sampled using nearest neighbor. If the cache contains a shaded value, it is immediately returned and used. If the cache location is empty, the pixel shader is invoked and the result put into the lowest level of the pyramid and propagated upwards. During upward propagation, the higher levels of the pyramid are sampled at the same uv-coordinates. If the higher level sample is empty, the same cached value is put into that pyramid level, and propagation continues. If the higher level sample is set, the propagation is aborted. When multiple threads operate on the same shading cache, sampling and propagation are subject to race conditions. We use atomic writes to the shading cache which makes it consistent, even during propagation. Without a shading cache, our system will perform far less shading requests than our stochastic rasterizer. The shading cache was added both for improving performance and to make the comparison with stochastic rasterization more fair.