Analytical Motion Blur Rasterization with Compression

Carl Johan Gribel[1], Michael Doggett[1] and Tomas Akenine-Möller[1,2]

[1]Lund University [2]Intel Corporation

Abstract

We present a rasterizer, based on time-dependent edge equations, that computes analytical visibility in order to render accurate motion blur. The theory for doing the computations in a rasterization framework is derived in detail, and then implemented. To keep the frame buffer requirements low, we also present a new oracle-based compression algorithm for the time intervals. Our results are promising in that high quality motion blurred scenes can be rendered using a rasterizer with rather low memory requirements. Our resulting images contain motion blur for both opaque and transparent objects.

1. Introduction

Motion blur is generated when the shutter of the camera is open for a finite time, and some relative motion appears inside the field of view of the camera. It is an effect that is important for offline rendering for feature films, since the frame rate is rather low ($\sim$ 24 frames per second). With motion blur in the rendered images, apparent jerkiness in the animation can be reduced or removed entirely. However, motion blur is also becoming an important visual effect for real-time rendering, e.g., for games. In order to get good performance, various rather crude approximations, that may or may not apply in all cases, are used.

In general, motion blur rendering can be divided into two parts, namely, *visibility determination* and *shader computations* [SPW02]. Most solutions that converge to a correctly rendered image are based on point sampling. The more samples that are used the better image is obtained, and at the same time, the rendering cost goes up. In many cases, one can obtain reasonable results with rather few shader samples compared to the number of visibility samples. For example, RenderMan uses only a single shader sample for motion-blurred micro polygons [CCC87, AG00].

With sampling comes noise. In the 1980's, a large amount of research was devoted to solving the motion blur visibility problem analytically [KB83, Cat84, Gra85] in order to avoid the noise. In our research, we fall back into this track, and again explore the possiblities of solving the visibility problem analytically. In particular, we explore rasterization of motion-blurred triangles with analytical visibility. In the next section, we review the most relevant previous work. This is followed by an edge equation primer in Section 3. A large part of this paper is then devoted to developing theory about visibility for moving triangles (Section 4). We present practical rendering techniques, including compression, in Section 5, which is followed by results and conclusions.

2. Previous Work

In this section, we review the most relevant previous work on motion blur rendering and visibility. In general, post-processing techniques have no chance of converging to the correct result. We aim to compute correctly converging motion blur for object and camera motion, which is a complex problem as pointed out by Vlachos [Vla08], and therefore, we avoid referencing post-processing techniques further. For general information about motion blur and for a great survey on the topic, we refer to Sung et al.'s work [SPW02].

In the 1980's, there were several papers on analytical visibility. Korein and Badler [KB83] describe two algorithms for motion blur rendering. The first is basically an accumulation buffering technique, where several images, rendered at different points in time, are weighted together. This type of algorithm has later been incorporated into graphics hardware [DWS*88, HA90] and graphics APIs. When only a few samples are used, this gives clear strobing artifacts, and for many samples, it becomes prohibitively expensive. In contrast, the second algorithm proposed by Korein and Badler is based on analytical visibility for motion blur. For each pixel (or more precisely, sample), a time interval when a disk is covering the sample is analytically computed. This is done for all objects and samples, and then they perform hidden surface removal, and resolve for the final sample color. In addition, they mention in an appendix that this is possible to do for polygons, and show how the equation of a polygon edge is set equal to the equation of a ray. However, no further details are given; the solutions are not derived, and there

is no depth function derivation either. Our work extends Korein and Badler's pioneering research in order to fill in these gaps.

Catmull [Cat84] takes another approach, where he argues that instead of using a stretched filter in the motion direction, he instead shrinks the object in the motion direction, and uses the same filter regardless of motion. The temporal filtering is thus turned into spatial filtering. This algorithm cannot handle moving objects that intersect other stationary objects. Grant [Gra85] develops an analytical visibility algorithm by treating moving three-dimensional polyhedra as static four-dimensional polyhedra. Unfortunately, the sides of the static polyhedra need to be planar, which greatly limits the usefulness of the algorithm. An example of when this is *not* true can be seen in Section 4.

Cook et al. [CCC87] presented the highly influential REYES rendering architecture. The geometric primitives are split, and then diced until they reach subpixel size. Shading is computed, and the visibility is sampled in both the temporal and spatial domains. This can generate some artifacts due to too little shading sampling in the temporal domain, but the temporal sampling rate can be increased by the user when needed. A stochastic rasterizer, targeting, but not limited to, low sampling rates is presented by Akenine-Möller et al. [AMMH07]. This work introduced time-dependent edge-functions, shadow maps, and environment maps as well. Ragan-Kelley et al. [RKLC*10] present a hardware architecture for motion blur and depth of field that separates shading from sampling, similar to that presented by Cook et. al. [CCC87]. While they also use time-continuous edge functions for motion blur, they sample them, unlike in this paper where we compute the analytical solution.

Sung et al. [SPW02] separate visibility and shading calculations, and perform analytical visibility using Korein and Badler's approach [KB83]. However, no details are given on how to actually perform the analytical visibility calculations. In contrast, we present all the mathematical details, and we do this in a rasterization framework in order to factor out per-triangle calculations in a triangle setup. Other contributions of this paper are the practical requirements for analytical motion blur such as compression of the many depth intervals that must be stored at each pixel.

Fatahalian et. al. [FLB*09] present an analysis of data parallel rasterization for micro polygons for motion blur and depth of field. They sample in the time domain and do not attempt to analytically calculate visibility.

3. Edge Equation Primer

In this section, we briefly present the necessary background on edge equations [Pin88, OG97] and time-dependent edge equations [AMMH07] for moving triangles.

Assume that the entire transform matrix, including projection, for a vertex is called $\mathbf{M}$. A vertex in homogeneous clip space is then obtained as $\mathbf{p} = \mathbf{M}\bar{\mathbf{p}}$, where $\bar{\mathbf{p}}$ is the vertex in three-dimensional object space, and $\mathbf{p}$ is the resulting four-dimensional vertex in homogeneous clip space, i.e., before division by the w-component. To simplify the derivation, we use the following notation: $\hat{\mathbf{p}} = (p_x, p_y, p_w)$, which is a scaled and translated version of the point in camera space. This can be confirmed by looking at the definition of the projection matrix in OpenGL and DirectX.

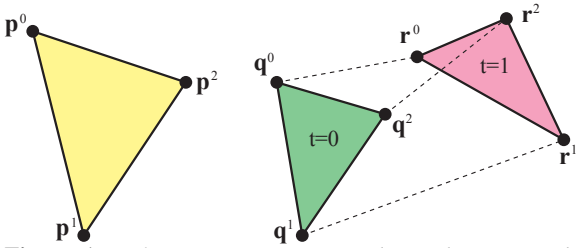


Figure 1: *Left: a non-moving triangle. Right: a triangle moving from time $t = 0$ to $t = 1$.*

The standard (no motion) edge function, in homogeneous form, through two vertices, say $\hat{\mathbf{p}}^0$ and $\hat{\mathbf{p}}^1$, is [OG97, MWM02]:

$$e(x,y,w) = (\hat{\mathbf{p}}^1 \times \hat{\mathbf{p}}^0) \cdot (x,y,w) = ax + by + cw. \quad (1)$$

A sampling point, (x,y,w), is inside the triangle if $e_i(x,y,w) \leq 0$ for $i \in [0,1,2]$, i.e., for the three edges of the triangle. Next, this is extended with a time dimension.

Assume that the vertices move linearly from the beginning of a frame, at $t = 0$, to the end for a frame, at $t = 1$. At $t = 0$, we denote the vertices as $\mathbf{q}^i$, and we call them $\mathbf{r}^i$ at $t = 1$. Since there is no bar nor a hat on the vertices, all $\mathbf{q}^i$ and $\mathbf{r}^i$ are in homogeneous clip space. A linearly interpolated vertex is then given as:

$$\mathbf{p}^i(t) = (1-t)\mathbf{q}^i + t\mathbf{r}^i, \quad (2)$$

for a certain instant $t \in [0,1]$. See Figure 1 for an example.

The coefficients of a time-dependent edge equation are given by [AMMH07]:

$$\begin{aligned}(a,b,c) = (\hat{\mathbf{p}}^1 \times \hat{\mathbf{p}}^0) &= ((1-t)\hat{\mathbf{q}}^1 + t\hat{\mathbf{r}}^1) \times ((1-t)\hat{\mathbf{q}}^0 + t\hat{\mathbf{r}}^0) \\ &= t^2\mathbf{f} + t\mathbf{g} + \mathbf{h}, \end{aligned} \quad (3)$$

where:

$$\begin{aligned}\mathbf{h} &= \hat{\mathbf{q}}^1 \times \hat{\mathbf{q}}^0, \\ \mathbf{k} &= \hat{\mathbf{q}}^1 \times \hat{\mathbf{r}}^0 + \hat{\mathbf{r}}^1 \times \hat{\mathbf{q}}^0, \\ \mathbf{f} &= \mathbf{h} - \mathbf{k} + \hat{\mathbf{r}}^1 \times \hat{\mathbf{r}}^0, \\ \mathbf{g} &= -2\mathbf{h} + \mathbf{k}. \end{aligned} \quad (4)$$

Each edge equation is now a function of time consisting of three functions: $(a(t), b(t), c(t))$, where, for example, $a(t) = f_x t^2 + g_x t + h_x$. Finally, the entire time-dependent edge equation is:

$$e(x,y,t) = a(t)x + b(t)y + c(t), \quad (5)$$

where we have set $w = 1$ since rasterization is done in screen space (x,y).

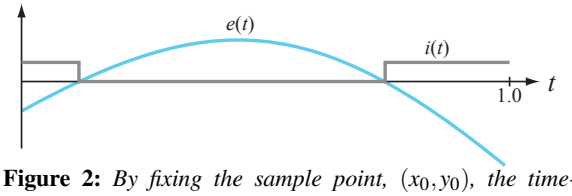


Figure 2: *By fixing the sample point, (x_0, y_0), the time-dependent edge function, $e(x,y,t)$, becomes a quadratic polynomial in t alone, i.e., $e(t)$. The blue curve is an example of that. In addition, a binary inside function, $i(t)$, is shown (green), which is true whenever $e(t) \leq 0$.*

4. Theory

In this section, we derive and present some new theory about time-dependent edge equations that we have not seen presented elsewhere.

4.1. Analytic Inside Test

For now, we assume that each pixel has a single sample point at (x_0, y_0). Extensions to multi-sampling and super-sampling just increase the sampling rate. If we consider a particular pixel, then (x_0, y_0) are constant. In this case, the time-dependent edge function becomes a function of time, t, alone:

$$e(x_0, y_0, t) = e(t) = a(t)x_0 + b(t)y_0 + c(t). \tag{6}$$

This expression can be expanded using Equation 3:

$$\begin{aligned} e(t) &= t^2(f_x x_0 + f_y y_0 + f_z) + t(g_x x_0 + g_y y_0 + g_z) \\ &\quad + (h_x x_0 + h_y y_0 + h_z) \\ &= \alpha t^2 + \beta t + \gamma, \end{aligned} \tag{7}$$

where (α, β, γ) are constants for a certain sample point, (x_0, y_0). Hence, each edge equation is a quadratic function in t. An example is shown in Figure 2. Next, we introduce a binary *inside*-function, $i(t)$, as:

$$i(t) = \begin{cases} 1, & \text{when } e(t) \leq 0 \\ 0, & \text{elsewhere,} \end{cases} \tag{8}$$

i.e., $i(t) = 1$ for all $t \in [0,1]$ when (x_0, y_0) is inside the corresponding time-dependent edge equation. Note that the inside functions, $i_k(t)$, can be computed analytically by solving the second-degree polynomial in Equation 7.

For a moving triangle, we have three time-dependent edge functions, $e_k(t)$, where $k \in \{0,1,2\}$. The point, (x_0, y_0), will be inside the moving triangle when all inside functions are positive at the same time. This visibility function can be expressed as:

$$v(t) = i_0(t) i_1(t) i_2(t), \tag{9}$$

i.e., the multiplication of all three inside functions. An example of this is shown in Figure 3.

4.2. Analytic Time-Dependent Depth Function

In this section, we will derive the equation for the depth during the time span where the sample point, (x_0, y_0), is inside the moving triangle. Perspective-correct interpolation coordinates, (u, v), can be used to interpolate any attribute per vertex. This is done as:

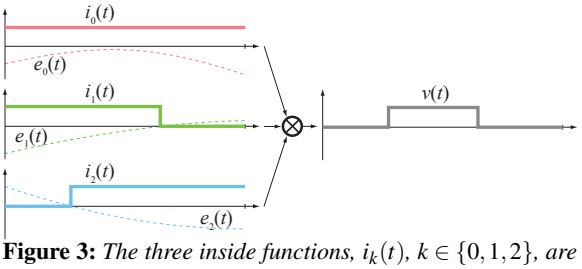


Figure 3: *The three inside functions, $i_k(t)$, $k \in \{0,1,2\}$, are shown to the left, and the resulting visibility function, $v(t)$, shown to the right, is positive for all values of t where all $i_k(t)$ are positive. Hence, the positive interval to the right is the time span where the sample point, (x_0, y_0), is inside the moving triangle.*

$$\mathbf{s}(u,v) = (1-u-v)\mathbf{p}^0 + u\mathbf{p}^1 + v\mathbf{p}^2, \tag{10}$$

where $\mathbf{p}^k$ are the attribute vectors at the three vertices, and $\mathbf{s}(u,v)$ is the interpolated attribute vector. McCool et al. [MWM02] showed that (u,v) can be computed using edge equations, e_k:

$$(u,v) = \frac{1}{e_0 + e_1 + e_2}(e_1, e_2). \tag{11}$$

Note that u, v, and all e_k are functions of (x_0, y_0), but this was left out to shorten notation. Equation 11 also holds when time-dependent edge equations are used.

The depth buffer stores interpolated depth values. Assuming that $\mathbf{p}^k = (p_x^k, p_y^k, p_z^k, p_w^k)$, $k \in \{0,1,2\}$, are the triangle vertices in clip space (before division by w), one first uses Equation 10, and then compute the depth as $d = s_z/s_w$ [SA06] for a particular fragment with perspective-correct barycentric coordinates, (u,v).

When we turn from static triangles to moving triangles, $\mathbf{p}^k$ are functions of time (Equation 2), and so are the edge equations. Let us first take a look at one of the texture coordinates, u (see Equation 11):

$$\begin{aligned} u &= \frac{e_1}{e_0 + e_1 + e_2} \\ &= \frac{\alpha_1 t^2 + \beta_1 t + \gamma_1}{(\alpha_0 + \alpha_1 + \alpha_2)t^2 + (\beta_0 + \beta_1 + \beta_2)t + \gamma_0 + \gamma_1 + \gamma_2}, \end{aligned} \tag{12}$$

where the three time-dependent edge equations are: $e_k(t) = \alpha_k t^2 + \beta_k t + \gamma_k$ (Equation 7). As can be seen, the texture coordinate, u, becomes a rational polynomial of degree two in t. The major difference, when compared to static triangles, is when the entire depth function is put together:

$$d(t) = \frac{s_z}{s_w} = \frac{(1-u-v)p_z^0 + up_z^1 + vp_z^2}{(1-u-v)p_w^0 + up_w^1 + vp_w^2}, \tag{13}$$

where all p_z^i and p_w^i are functions of time according to Equation 2, and u & v are functions of time (Equation 12) as well. When these expressions replace the corresponding terms in

Equation 13, we arrive at a cubic rational polynomial for the depth function for a certain sample point, (x_0, y_0):

$$d(t) = \frac{m_3 t^3 + m_2 t^2 + m_1 t + m_0}{n_3 t^3 + n_2 t^2 + n_1 t + n_0}, \tag{14}$$

where the constants in this equation, typically, are rather long and complex expressions.

Two of the advantages of using $d = s_z/s_w$ includes the fact that the depth is in the range $[0,1]$ due to the way the projection matrix is set up and that depth buffer compression can therefore be implemented efficiently since the depth will be linear over a triangle. In our rendering framework, we will not be able to use depth compression anyway, so instead we can use $d = s_z$, which will generate the same images, but the depth will now range between the near and the far plane: $[z_{near}, z_{far}]$. This simplifies the depth function for moving triangles. It will still be a rational function in t with degree three in the numerator, but the degree in the denominator will be reduced to two, that is:

$$d(t) = \frac{m_3 t^3 + m_2 t^2 + m_1 t + m_0}{n_2 t^2 + n_1 t + n_0}, \tag{15}$$

In Section 5, we will approximate the exact depth described by Equation 15 with a linear depth function.

4.3. Visualization of Moving Triangles

Each moving edge traces out a "side surface," and these form bilinear patches. In this section, we attempt to help to build some intuition about moving triangles, their visibility functions (Equation 9), and depth functions (Equation 15). An example of a moving triangle visualized in three different ways is shown in Figure 4. Note that in the middle figure, the gray curve is actually the visibility function multiplied by the depth function that is shown.

It is also interesting to note that there can be several disjoint parts of the visibility function (Equation 9). Examples with two and three disjoint parts, and their respective depth functions, are shown in Figure 5.

In theory, the intersection of three inside functions of the visibility function can result in at most four disjoint time spans where the resulting function, $v(t)$, is positive. This is because each inside function can consist of two disjoint positive parts. In practice, we have only encountered three intervals when you consider *front-facing* triangles for any value of t. Most commonly, only a single interval is generated for most triangles and samples, however.

5. Rendering

In the following, we will use the term *interval* to denote a range in the time dimension, together with the color and depth for that time range. An interval is denoted by Δ. In practice, the third-degree rational depth function (Equation 15), is approximated by a linear function. The motivation for this is that the depth function rarely varies much

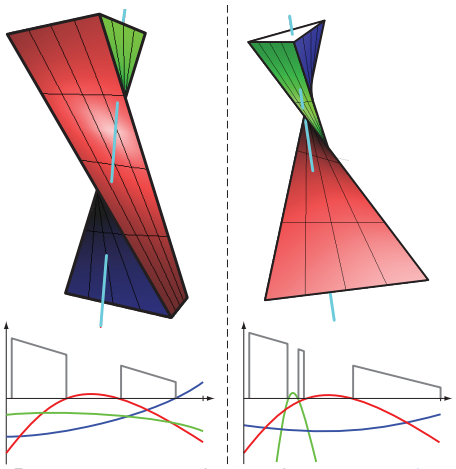


Figure 5: *Two more examples, similar to Figure 4. Left: the sample point (illustrated by the cyan line) is inside the moving triangle in two disjoint time intervals as can be seen by the depth function below. Right: the sample point is inside in three disjoint time intervals.*

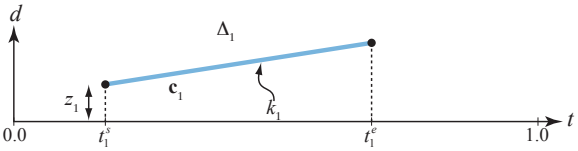


Figure 6: *Here we illustrate the parameters for an interval, Δ_1, which is defined by its starting and ending point in time, t_1^s and t_1^e, its depth, z_1, at t_1^s, the slope, k_1, of the depth function, and the color, $\mathbf{c}_1$, of the interval.*

beyond such an approximation, and it makes computations much faster. In addition, we have good experiences with this approximation, as will be shown in Section 6.

Given the depth function approximation, each interval stores the following parameters:

$$\begin{aligned} t_i^s &: \text{time at the beginning of the interval} \\ t_i^e &: \text{time at the end of the interval} \\ z_i &: \text{depth at the beginning of the interval} \\ k_i &: \text{slope of the depth function} \\ \mathbf{c}_i &: \text{color of the interval} \end{aligned} \tag{16}$$

Our interval is analogous to a fragment in rendering without motion blur, and an example of an interval is shown in Figure 6. In general, all intervals belonging to a pixel are simply stored in an interval list in that pixel, similar to how order-independent transparency is done with DX11 using current GPUs [Mic09]. A couple of optimizations that can be used for opaque rendering will be described below. As a triangle that covers a pixel is rendered, one or more intervals are added to that pixel's interval list. When rendering starts, each pixel is initialized with an interval with background color and depth equal to the far plane.

Interval insertion is illustrated in Figure 7. Our approach is

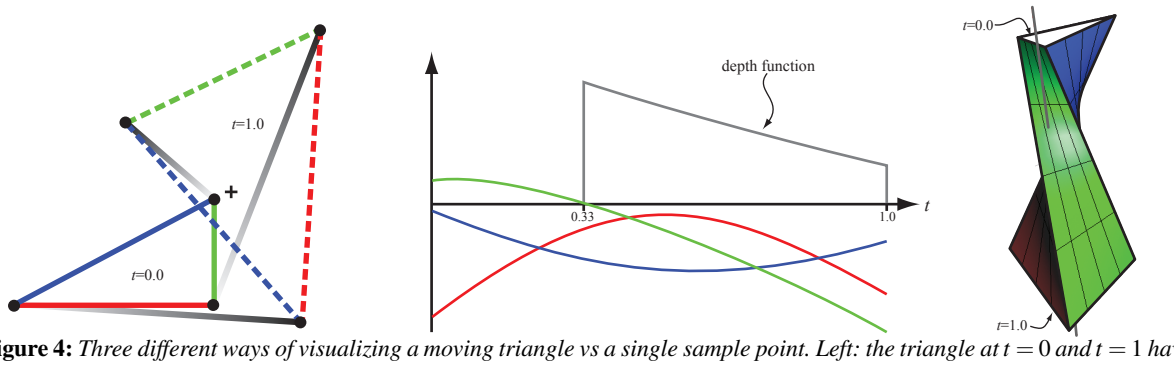


Figure 4: *Three different ways of visualizing a moving triangle vs a single sample point. Left: the triangle at $t = 0$ and $t = 1$ have been projected to screen space. Note that the lines with white-to-black gradients depict the trajectories of the vertices. At $t = 1$, the triangle is shown with dashed lines. Middle: the edge equations for the three edges are shown with colors corresponding to the colors to the left. All curves here are for the sample point indicated by a cross to the left. Note that the sample point is inside the moving triangle from $t = 0.33$ to $t = 1$, and that the depth on the triangle during this time is depicted as the gray depth function. Right: The triangle vertices, (x, y, w), have been rendered in three dimensions with the "side surfaces" of each moving edge rendered as a bilinear patch. The edges are color coded in the same way as in the left and middle image. The sample point is depicted as a gray line here. Note that the gray line does not intersect the moving triangle to start with, and that it then enters into the green surface. This is illustrated in the middle figure when the green edge equation curve becomes negative at $t = 0.33$.*

based on trying to keep the number of intervals stored per pixel small, and to facilitate compression (discussed in Section 5.2) when possible. When two intervals intersect, one can use clipping to create non-overlapping (in time) intervals. However, that can generate up to four intervals, which is undesirable. An intersection can also generate only two intervals, but in such cases, we also refrain from clipping since our compression mechanism works better with unclipped intervals. Note that using non-clipped intervals requires a slightly more complex resolve procedure, which is described in Section 5.1. For *opaque* rendering, simple depth test optimizations can be included in this process as well, and this is shown in the bottom two illustations of Figure 7, where we have assumed that a `LESS_EQUAL` depth test is used. It is, however, straightforward to adapt to any depth function. In the second illustration from the bottom, the green interval is clipped in time since it is occluded and because the clipping only generate one interval, i.e., it does not require more storage than simply storing the unclipped interval.

Note that to facilitate depth testing, we keep the intervals sorted on t_s^i per pixel. This can be done during interval insertion using insertion sort, for example. In the next subsection, we will describe how the final color of each pixel is resolved given such a list of intervals. This is followed by Section 5.2, where we describe a novel compression algorithm for the interval list stored per pixel.

5.1. Temporal Pixel Resolve

After rendering all moving and non-moving triangles, we have a list of possibly overlapping (in time) intervals, Δ_i, per pixel. Recall that we keep the intervals sorted, i.e., $t_s^i \leq t_s^{i+1}$, $\forall i$, since this is part of the interval insertion step described above.

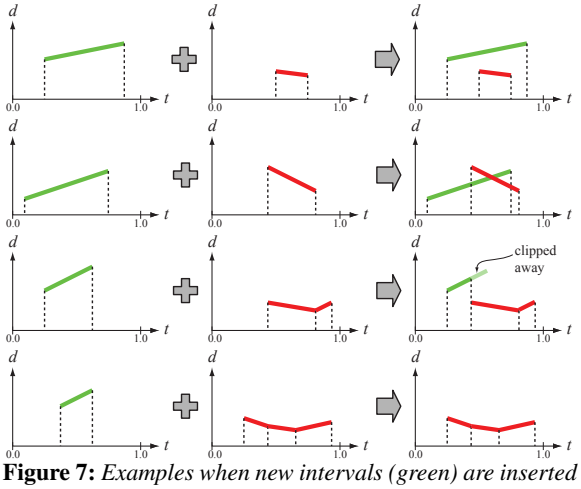


Figure 7: *Examples when new intervals (green) are inserted into the existing interval list (red). At the top, clipping the green interval would result in the existing red interval and two green intervals, one on each side of the red interval. Our approach is to simply store the entire green interval, since that requires only two intervals in total (compared to three). Second from top: for intersections, the new interval is also added to the pixel's interval list. The bottom two images show optimizations which are valid only for opaque rendering with a `LESS_EQUAL` deph test.*

The resolve pass processes a pixel independently of other pixels, and sweeps the sorted intervals in a pixel from time $t = 0$ to $t = 1$. During the sweep, we maintain a list, called *Active List*, of all intervals overlapping the current time of the sweep. For example, when the sweep starts, at $t = 0$, the *Active List* is initialized to hold all intervals that overlap $t = 0$. Note that the *Active List* is also kept sorted on depth. As an interval start-point, t_i^s, or end-point, t_i^e, or an

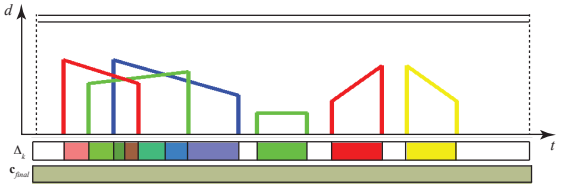


Figure 8: *Motion blur of semi-transparent geometry. Each interval represents exposure from a primitive, whereas the topmost interval represent a white background. The trio to the left have α set to 0.6 and the trio to the right have α set to 1.0. The multi-colored bar shows the alpha-blended colors for each respective subspan in time, while the bar at the bottom shows the final color.*

intersection point between two intervals is encountered, the following action is taken:

- Interval-Start: insertion-sort (on depth) new interval into *Active List*.
- Interval-End: remove interval from *Active List*.
- Interval-Intersection: swap places of the intersecting intervals in the *Active List* to maintain depth order.

Between each pair of encounters (the three cases above), the final color for that particular subspan in time is computed. Each such color and subspan in time are temporarily put into a *resolved interval*, Δ_k. Since intersection points are handled as part of the sweep, there will not be any intervals overlapping in depth in the *Active List*. As a consequence of that, the color of Δ_k for *opaque* rendering is simply the color of the nearest (in depth) interval in the *Active List*, again assuming that a `LESS_EQUAL` depth test is used. Each *resolved* interval's color, $\mathbf{c}_k$, is then integrated against the shutter response function, $w(t)$, and added to the final color of the pixel. This can be expressed as:

$$\mathbf{c}_{final} = \sum_{k=0}^{n-1} \left(\int_{t_k^s}^{t_k^e} w(t)\mathbf{c}_k dt \right), \tag{17}$$

for n disjoint intervals. If a box filter is used, the colors of all intervals are simply combined into one final color weighted by their duration in time.

For the transparent resolve procedure, the only difference is that the color, $\mathbf{c}_k$, of the resolved interval, Δ_k, is computed by blending the intervals in the *Active List* in back-to-front order based on the alpha component of each color [PD84]. An example is shown in Figure 8.

5.2. Temporal Pixel Compression

When there are many small triangles with a relatively high degree of motion blur, each pixel may need to store a large number of intervals, Δ_i, in order to exactly represent the color of the pixel. We have observed up to a few hundred intervals per pixel in extreme cases. This is clearly not desirable for a rasterization-based algorithm. The problem can be alleviated by using a tiling architecture [FPE*89], where the

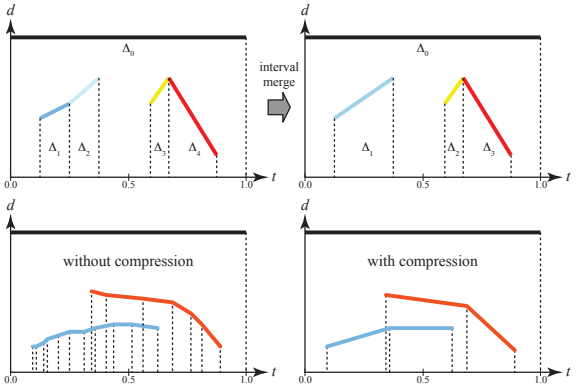


Figure 9: *Top: illustration of the merging of two intervals. Since the bluish intervals have similar depth functions, they are deemed suitable for merging by our oracle. Bottom left: illustration of content in a pixel after rendering many small moving triangles. Bottom right: a possible result when using our compression algorithm. Note that our compression algorithm is applied continuously as each moving triangle is being rendered, and in that sense, our technique is similar to, for example, standard depth buffer compression.*

triangles are sorted into tiles (rectangular regions of pixels) by front-end pass, and per-pixel rendering done in a back-end pass. Tiles can be back-end processed in parallel by separate cores, since the rasterization and per-pixel work is independent at this point. Note that even the XBOX 360 reverts to tiling when multi-sampling anti-aliasing and high resolutions are used. Even if tiling is used, storage may still be a problem since the per-pixel storage is still unbounded. However, we note that there is lots of coherence to exploit in this problem, and this section is therefore dedicated to introducing a per-pixel lossy compression algorithm.

The core idea is illustrated in the top part of Figure 9. Assume that a pixel can only "afford" to store four intervals, and that after rendering another motion blurred triangle, the pixel actually holds five intervals (top left in Figure 9). To be able to fit this into our frame buffer, we will need to compress this information into four intervals again. This is shown in the top right part of the figure, where the two bluish intervals with similar depth functions have been merged to a single interval. A similar type of compression has been used for deep shadow maps [LV00], but in their context, compression was only needed to be done once after the entire visibility function was known. Our goal is different since we may need to compress each pixel several times as more and more triangles are rendered to a pixel.

We use an oracle-based approach to attack this problem. Our oracle function is denoted:

$$o_{ij} = O(\Delta_i, \Delta_j), \tag{18}$$

where the oracle function, $O()$, operates on two intervals, Δ_i and Δ_j, where $i < j$. The task of the oracle is basically to compute an estimation on how appropriate it is to

merge the two input intervals. Given an oracle function, O, we compute oracle function values, o_{ij}, for all i, and $j \in \{i+1, i+2, i+3\}$. For transparent scenes with high level of depth overlap, we have increased the search range up to $+10$, instead of $+3$. In the next step, the interval pair with the lowest o_{ij} is merged. Depending on the implementation, this process may continue until the number of intervals per pixel falls in a desired range, or until there are no more appropriate merges possible. Next, we describe an oracle, for two intervals, that generates a lower value the more appropriate they are to merge.

5.2.1. Oracle Function

Our oracle function, O, is described by the following formula, where $i < j$:

$$\begin{aligned} O(\Delta_i, \Delta_j) = {} & h_1 \max(t_j^s - t_i^e, 0) + h_2 |\bar{z}_i - z_j| + h_3 |k_i - k_j| + \\ & h_4 (t_i^e - t_i^s + t_j^e - t_j^s) + \\ & h_5 (|c_{i,r} - c_{j,r}| + |c_{i,g} - c_{j,g}| + |c_{i,b} - c_{j,b}|). \end{aligned} \quad (19)$$

The first term favors merging of intervals that are located close in time (even overlapping). For the second term, $\bar{z}_i$ is the depth at the end of Δ_i, i.e., $\bar{z}_i = z_i + k_i(t_i^e - t_i^s)$, and z_j is the depth at the beginning of the other interval, Δ_j. See Equation 16 for definitions of the interval's parameters. The third term penalizes merging of intervals with different slopes. Hence, both the second and third terms attempt to detect if the depth functions are similar, and therefore, whether they are amenable for merging. The fourth term favors merging of short (in time) intervals, while the fifth favors merging of interval with similar colors. All h_i are used-specified constants, and we will discuss our values of these constants in Section 6.

Discussion Choosing an oracle function is always a rather ad-hoc process. Here, we will present the methodology that we used in doing this, and some of the attempts that failed. First of all, we implemented a *temporal pixel visualizer*, where we could visually inspect all temporal fragments that were added to a pixel's interval list, and what the compressed representation looked before and after. In the end, what the temporal pixel visualizer revealed to us was that the depth functions should be similar, and that the intervals should be located closely in time. In addition, for small triangles, we saw that shorter intervals are easier to merge, and that we got better image quality if we favor merging of intervals with similar color. Furthermore, we tried an approach for opaque rendering, where all intervals with intersecting depth functions were clipped. This resulted in much worse quality for lower number of intervals per pixel, but worked equally well as the method we presented above, when there was a high degree of occlusion in the scenes. We also considered introducing an error term per interval in order to avoid *tandem compression*, i.e., where one interval gets compressed several times with the result being that the merged interval has little resemblance to the original intervals. If an interval has been formed by merging two previous intervals, and a further interval attempts to merge with it, the error term would penalize the merging with that new interval. However, in our test scenes we have not seen any need for this (by examining compression artifacts in our temporal pixel visualizer) so far. For other test scenes, this may pay off, though, so we hope to investigate this in future work.

5.2.2. Interval Merging

Our merging process for two intervals is straightforward. We will describe the merging of two intervals, Δ_i and Δ_j, into a new interval, Δ_i'. The merge is described as: $\Delta_i' = \texttt{merge}(\Delta_i, \Delta_j)$, $i < j$, where the new parameters are:

$$\begin{aligned} t_i'^s &= t_i^s \\ t_i'^e &= \max(t_i^e, t_j^e) \\ z_i' &= (1-\alpha) z_i + \alpha (z_j - k_j (t_j^s - t_i^s)) \\ k_i' &= (1-\alpha) k_i + \alpha k_j \\ \mathbf{c}_i' &= (1-\alpha) \mathbf{c}_i + \alpha \mathbf{c}_j, \end{aligned} \quad (20)$$

where $\alpha = (t_j^e - t_j^s)/(t_i^e - t_i^s + t_j^e - t_j^s)$ is used to linearly blend parameters depending on the lengths (in time) of the intervals that are merged. As can be seen, the slope, k_i', of the depth function, and the color, $\mathbf{c}_i'$, are simply linear interpolations of the input intervals' parameters. The depth, z_i', is slightly more complex because we need to blend the depth at the same instant in time. Since we want the new depth at time t_i^s, we compute the depth of Δ_j's depth function at t_i^s and use that for blending. For future work, it would be interesting to investigate other approaches to depth merging, e.g., where the area under the depth function is kept constant after compression. An example of merging two intervals is shown in Figure 9.

6. Results

All algorithms described in the paper have been implemented and tested in a rendering test framework in C++, where the core is a software rasterizer. We have implemented stochastic rasterization to generate comparison images, and ground truth images (with 256 samples). To generate compelling animations, we have integrated the Bullet physics engine into our framework.

For our oracle (Equation 19), we use the following constants in all renderings: $h_1 = 1000$, $h_2 = 100$, $h_3 = 1$, $h_4 = 100$, and $h_5 = 1$. This setup has worked well for all our test scenes, i.e., with different triangle sizes, occlusion, disparity in color, and different speeds of motion. More time could definitely be spent on tuning these parameters.

To motivate our approximation of the depth function, we gathered the maximum depth errors introduced for each interval. These depth errors were measured relative to the difference between the far and near planes, which were

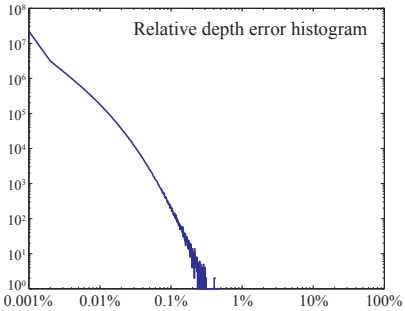


Figure 10: *Illustration of the error introduced by approximating the cubic rational depth function with a linear function. Note that we use logarithmic scale on both axes here.*

trimmed down. A histogram of the result is shown in Figure 10 for all intervals in 25 frames of the bunny scene (Figure 15). As can be seen, the vast majority of the area under the histogram function resides in the left area, which means that most intervals introduce only tiny errors. In fact, all intervals have less than 0.6% relative depth error, and 96.3% of the intervals have less than 0.01%. This implies that the approximation is reasonable.

It is interesting to compare the size of the frame buffers for our algorithms vs. stochastic rasterization. We assume that a sample stores RGBA in 32 bits, and depth in 32 bits. This sums to 8 bytes per sample for stochastic sampling. In our algorithm, an interval, Δ_i, also stores RGBA and depth using 8 bytes in addition to the start and end times, t_i^s & t_s^e, each using one byte, and the slope, k_i, as a 16-bit float. This sums to 12 bytes per interval. For the same frame buffer memory usage, stochastic rasterization can use, e.g., 12 samples per pixel, while we use eight intervals. Hence, the overhead for our algorithm is 50% in terms of memory storage. It should be noted here though that stochastic sampling does address spatial anti-aliasing as well, whereas our algorithm does not.

Figure 13 and 14 show rendering of opaque surfaces. Both figures show that the analytical evaluation generates accurate and smoothly blurred images practically free of the sampling noise typical of existing techniques with low numbers of samples. In Figure 14, the spheres are highly tessellated to approximate pixel size and shading is performed per triangle. To accommodate the large number of triangles, our compression technique uses 8 intervals per pixel in Figure 14.

Figure 15 shows the results of our compression algorithm with different numbers of intervals for a scene with fast motion. As can be seen, already at 8 intervals per pixel, our results are very hard to distinguish from the ground truth. We also gathered statistics on the number of inserted intervals for the bunny scene in Figure 15. For the entire animation, the average number of intervals per pixel was 7.2, while the maximum 177. This suggests that there may be other possible implementations, where all non-occluded intervals are stored without compression, and later resolved in a slightly more complex procedure. We leave this for future work.

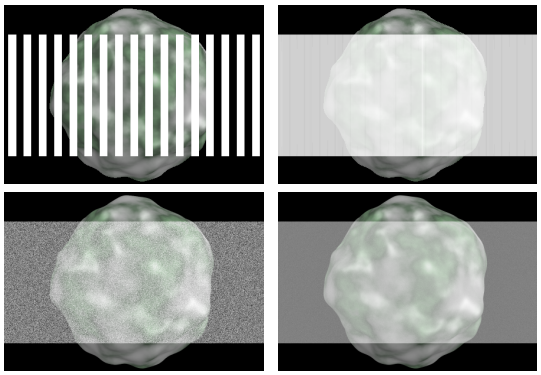
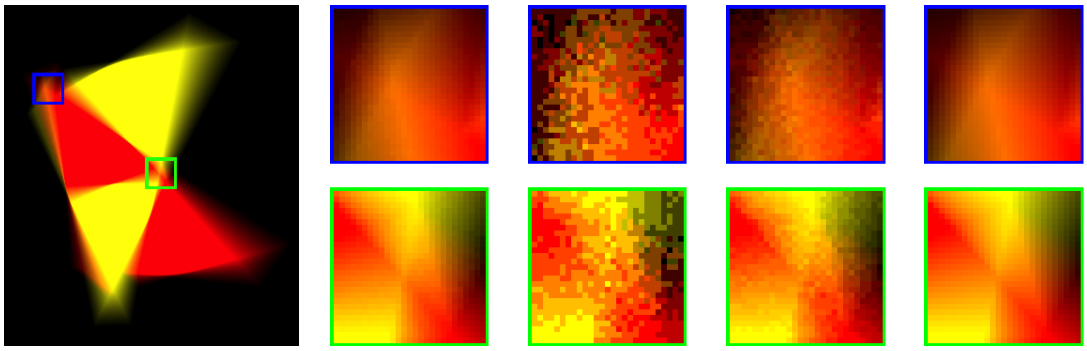


Figure 11: *A very difficult case for our compression algorithm. Top left: static rendering, where the white "fence" is moving to the left at a speed of **500** pixels per frame. Top right: our compression algorithm with 16 intervals per pixel, which fails at reproducing an accurate image. About 40 intervals are needed for our algorithm to create a high quality image. Bottom images: 24 samples (left) and 256 samples (right) with stochastic rasterization.*

Our non-optimized rasterizer, running on a single-threaded 3.2GHz Quad Core Intel Xeon, renders these frames at about 16 seconds per frame on average. It should be possible to speed up this CPU code by at least a factor of 10, and a GPU implementation can take us yet further.

As with most compression algorithms, one can create cases where they behave poorly. For our algorithm, this can occur for *extreme* motion, and we have rendered one such example in Figure 11. The reason why our compression algorithm fails at handling this case accurately is that while rendering the fences, the pixel will try to store a visibility function that resembles a square wave function. Compressing such a function will either destroy the foreground (white) or the background (either black or marble stone), and so there is little chance of ending up at a perfect mix of 50% foreground and 50% background. Still, we believe that our compression algorithm behaves in a reasonable way given that it basically has pixel data storage only for an undersampled representation of the pixel content in this example. In addition, one can clearly see that our algorithm is only computing visibility in the time domain, and not spatially. This is another weakness of our current approach, and something that we will address in future work. Another interesting idea would be to replace the linear motion (Equation 2) with a higher order motion description.

Figure 12 shows our results for semi-transparent rendering. In general, many more intervals are needed when semi-transparency is used. For all images in this animation, each pixel was compressed to 64 intervals, and the maximum number of intervals being inserted in a single pixel for the entire animation was 122.



(a) (b) (c) (d) (e)

Figure 13: *Two intersecting triangles with zoom-ins outlined in blue and green. (a) Ground truth full image using 256 samples per pixel (SPP) with stochastic rasterization, (b) 256 SPP, (c) 4 SPP, (d) 16 SPP, and (e) our analytical motion blur rasterization.*

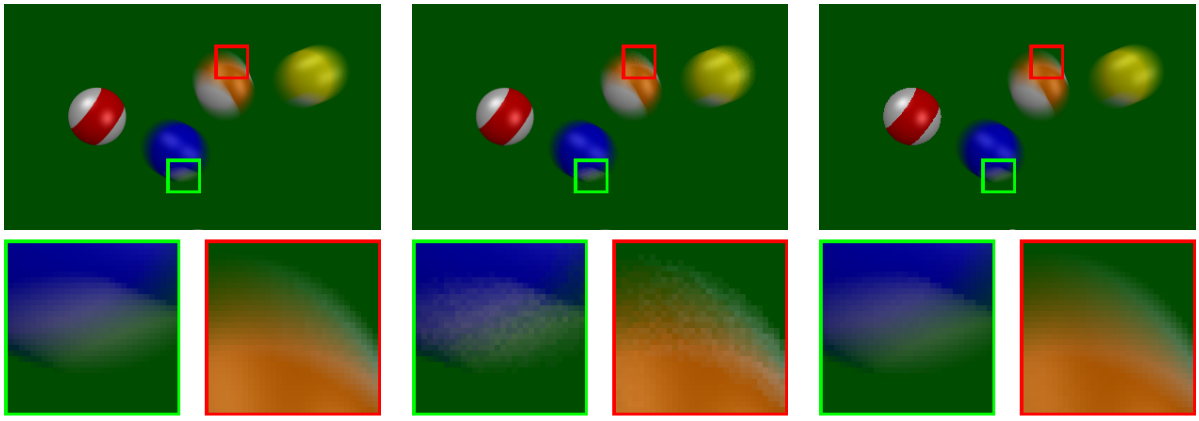


Figure 14: *Billard ball motion with zoom-ins outlined in red and green. Left: ground truth images with 256 samples per pixel (SPP). Middle: 12 SPP stochastic rasterization. Right: our analytical motion blur rasterization using 8 intervals for compression. Notice the absence of noise to the right.*

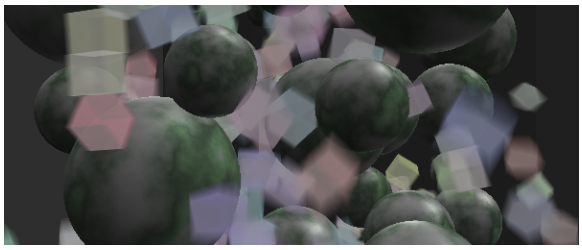
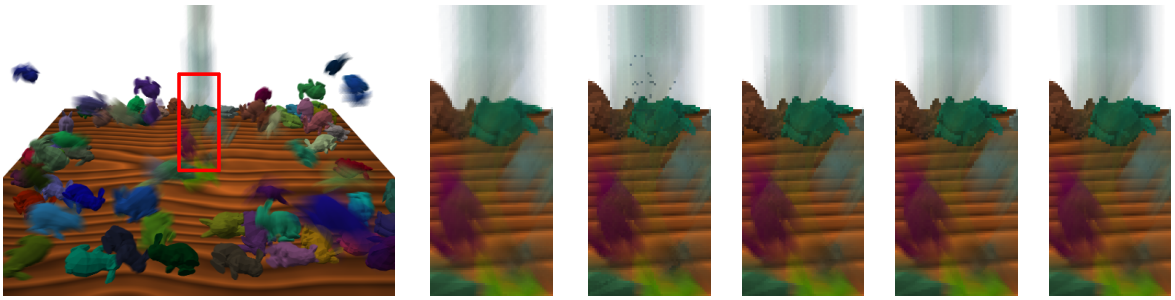


Figure 12: *A scene with moving semi-transparent cubes falling between spheres with a procedural marble shader demonstrates that our technique also works well with semi-transparency.*

7. Conclusions and Future Work

We have presented the mathematical details on analytical visibility computations for motion blurred triangles. Even though similar work appears to have been done before [KB83, SPW02], the details have been missing, or perhaps cruder approximations been used, and they have not been introduced in a rasterization framework. We hope that our work will spur renewed interest in analytical visibility computations, especially since the gap between compute power and memory bandwidth continues to increase. To make this approach more practical, we introduced some approximations, e.g., temporal pixel compression and a linear depth function approximation.

Admittedly, we have not yet implemented a fully practical rendering algorithm. As a result, there are many interesting avenues for future work around this topic. As a next step, we will implement our algorithms in DirectX ComputeShader or in OpenCL, so that the power of GPUs can be used to accelerate our algorithm. It is likely that this will lead the way to new insights that may improve the algorithm further in order to make it truly practical. The approximations that we introduced are one area that deserves more research. We wish to investigate whether there are other parts of the algorithms that are better suited for compression, and to look into whether the current approximations can be improved. Furthermore, we will study how shadow mapping can be adapted to our algorithms so that motion blurred shadows can be cast on moving objects. A major topic that also deserves more attention is shading. Currently, all triangles can



(a) (b) Reference (c) Rate 4 (d) Rate 6 (e) Rate 8 (f) Rate 10

Figure 15: *Compression at different rates. The image in (a) displays bunnies being dropped onto a table from high altitude. As can be seen by the washed out stream of motion blur at center, they travel at significant speed before impact. The highlighted area depicts a particularly busy zone containing motion as well as geometric overlap. At a compression rate of 4 intervals per pixel (c), the image quality is not sufficient. However, at rates of 6–10 intervals (d-f), or more, the algorithm is able to preserve more and more of the true nature of the exposure intervals, resulting in higher quality images. The reference image in (b) was made with stochastic rasterization using 256 samples per pixel.*

only have a single color. We believe that a solution based on Ragan-Kelley et al.'s [RKLC*10] shader cache can also work for us. Finally, it would be interesting to adapt our work in the context of ray tracing, and see what benefits can be obtained there. We see our work as a small step forward for analytical visibility research, but very useful to anyone that wants to implement analytical motion blur. Due to the compute/memory gap, we believe that this could also be beneficial to the graphics industry in the long term.

Acknowledgements We acknowledge support from the Swedish Research Council and the Swedish Foundation for Strategic Research. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

References

[AG00] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000. 1

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16. 2

[Cat84] CATMULL E.: An Analytic Visible Surface Algorithm for Independent Pixel Processing. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)* (1984), pp. 109–115. 1, 2

[CCC87] COOK R. L., CARPENTER L., CATMULL E.: The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)* (1987), pp. 96–102. 1, 2

[DWS*88] DEERING M., WINNER S., SCHEDIWY B., DUFFY C., HUNT N.: The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)* (1988), pp. 21–31. 1

[FLB*09] FATAHALIAN K., LUONG E., BOULOS S., AKELEY K., MARK W. R., HANRAHAN P.: Data-Parallel Rasterization of Micropolygons with Defocus and motion Blur. In *High Performance Graphics* (2009), pp. 59–68. 2

[FPE*89] FUCHS H., POULTON J., EYLES J., GREER T., GOLDFEATHER J., ELLSWORTH D., MOLNAR S., TURK G., TEBBS B., ISRAEL L.: Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System using Processor-Enhanced Memories. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)* (1989), vol. 23, pp. 79–88. 6

[Gra85] GRANT C. W.: Integrated Analytic Spatial and Temporal Anti-Aliasing for Polyhedra in 4-Space. In *Computer Graphics (Proceedings of ACM SIGGRAPH 85)* (1985), pp. 79–84. 1, 2

[HA90] HAEBERLI P., AKELEY K.: The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 90)* (1990), pp. 309–318. 1

[KB83] KOREIN J., BADLER N.: Temporal Anti-Aliasing in Computer Generated Animation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 83)* (1983), pp. 377–388. 1, 2, 9

[LV00] LOKOVIC T., VEACH E.: Deep Shadow Maps. In *Proceedings of ACM SIGGRAPH 2000* (2000), pp. 385–392. 6

[Mic09] MICROSOFT: *Order Independent Transparency*. DirectX 11 SDK Sample, 2009. 4

[MWM02] MCCOOL M. D., WALES C., MOULE K.: Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware* (2002), pp. 65–72. 2, 3

[OG97] OLANO M., GREER T.: Triangle Scan Conversion using 2D Homogeneous Coordinates. In *Workshop on Graphics Hardware* (1997), pp. 89–95. 2

[PD84] PORTER T., DUFF T.: Compositing Digital Images. In *Computer Graphics (Proceedings of ACM SIGGRAPH 84)* (1984), pp. 253–259. 6

[Pin88] PINEDA J.: A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)* (August 1988), ACM, pp. 17–20. 2

[RKLC*10] RAGAN-KELLEY J., LEHTINEN J., CHEN J., DOGGETT M., DURAND F.: *Decoupled Sampling for Real-Time Graphics Pipelines*. Tech. Rep. MIT-CSAIL-TR-2010-015, 2010. 2, 10

[SA06] SEGAL M., AKELEY K.: *The OpenGL Graphics System: A Specification*. Tech. Rep. 2.1, 2006. 3

[SPW02] SUNG K., PEARCE A., WANG C.: Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics, 8*, 2 (2002), 144–153. 1, 2, 9

[Vla08] VLACHOS A.: Post Processing in The Orange Box. In *Game Developers Conference* (2008). 1