# Analytical Motion Blurred Shadows

Johan Palmér

Thesis for a diploma in computer science, 30 ECTS credits,
Department of Computer Science, Faculty of Science, Lund University

Examensarbete för 30 hp,
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

**Analytical Motion Blurred Shadows**
**Abstract**

A rendering framework supporting analytical visibility is extended with shadow mapping. Shadow maps containing analytical visibility data are used, leading to cases where both the projections to the shadow map and the depth tests can be time-dependent. For receivers that are static with respect to the camera, the depth tests are solved analytically over time. For dynamic receivers, point sampling is used. Problems arising from time-dependence, limited precision and necessary simplifications are investigated, and potential solutions are discussed.

**Analytiska rörelseoskarpa skuggor**
**Sammanfattning**

En rastrerare med stöd för analytisk rörelseoskärpa integreras med *shadow mapping*. Shadow maps med analytisk synlighetsinformation används för detta, vilket leder till situationer där både projiceringarna till shadow map:en och djupjämförelserna kan vara tidsberoende. Djupjämförelserna utförs analytiskt över tiden för mottagare som är statiska i förhållande till kameran. För dynamiska mottagare används *point sampling* istället. Problem som uppstår på grund av tidsberoende, begränsad precision och nödvändiga förenklingar undersöks, och potentiella lösningar diskuteras.

# Contents

# Acknowledgements

I would like to thank my supervisor Tomas Akenine-Möller for helping me get started with this thesis and providing much help and encouragement along the way.

# Chapter 1

# Introduction

## 1.1 Computer Graphics

Computer graphics concerns the rendering of images on a computer. In three-dimensional computer graphics, a renderer is typically given a description of a scene, which it generates a two-dimensional image from. The scene can consist of a virtual camera (which represents the viewpoint to render from), geometrical objects, material data for these objects, light sources, etc.

For real-time applications, a *rasterization*-based renderer is most often used, due to its efficiency compared to other techniques such as ray tracing. Rasterization is the process of converting vector graphics to pixels. The standard algorithm for this kind of renderer is the *rendering pipeline* [1]. This pipeline consists of a number of stages that work in parallel, each one generating data that is given as input to the next stage. This means that the slowest stage, the bottleneck, determines the rendering speed. Each stage can also be parallelized.

In an application, the renderer is first provided with the scene description. Other processes, such as collision detection and user input, are usually performed as well. After that, the renderer is executed. The pipeline is usually implemented in hardware. Figure 1.1 shows an example of a rendering pipeline. The stages can vary depending on the implementation, but can be coarsely divided into two conceptual steps: geometry and pixel processing.

### 1.1.1 The Geometry Stage

The geometry stage performs operations per-polygon and per-vertex. A model first resides in *model space*, which means that the vertices are specified relative to the model. After the *model transform* has been applied to

the vertices, they are located in *world space*. From world space, the *view transform* is applied to transform the geometry to *eye space*.

Then a *projection transform* transforms the vertices into *clip space*. Often a perspective transform is used, which makes distant objects appear smaller.

These transforms are usually performed by executing a *vertex shader* for each vertex. This vertex shader can also compute other parameters (to be sent to the next stage in the pipeline), such as per-vertex lighting or animation.

The final step is to map the coordinates from clip space to screen space.

### 1.1.2 Pixel Processing

The rasterizer is given primitives in screen coordinates as input. For each primitive, the rasterizer determines which pixels on the screen the primitive overlaps and computes the color of each one. The color for each pixel is stored in the color buffer, which is a rectangular array of colors.

The rasterizer also has to resolve visibility; it must handle the case of overlapping primitives in screen space. If any previous values in the color buffer are directly overwritten for each primitive, the final value of a pixel will be the color belonging to the last rasterized primitive overlapping that pixel, regardless of which primitive is closest to the camera. To solve this, a *Z-buffer*, or *depth buffer* is normally used. This buffer has the same dimensions as the color buffer, and for each pixel stores the depth value of the primitive currently closest to the camera. When a primitive is rasterized for a certain pixel, a *depth test* is performed before the color is computed and written to the color buffer. The depth test compares the depth value of the current primitive with the value stored in the depth buffer at the same pixel. If the depth of the current primitive is less [1] than the value in the depth buffer, the current primitive is closer to the camera than the previously closest primitive at that pixel. In that case, the color for the pixel is computed and written to the color buffer, and the depth value for the primitive is written to the depth buffer. If the depth value of the primitive is greater than or equal to the value in the depth buffer, nothing is written to the color buffer or the depth buffer. The depth buffer allows (opaque[2]) primitives to be rendered in any order.

[1] The depth test can also often be set to a different comparison, such as greater-than.

[2] Semi-transparent primitives are outside the scope of this thesis. To render them correctly with a z-buffer storing only one depth value per pixel, they need to be rendered in back-to-front order, and after all opaque primitives.

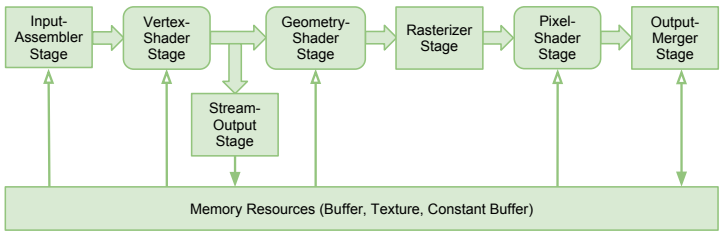



Figure 1.1: The rendering pipeline used in Direct3D 10 [4].

## 1.2 Motion Blur

Motion blur is a common effect in photography that appears when objects in an image move relative to the camera. This is because the image does not represent an instant of time, but the entire interval during which the shutter of the camera is open. For example, if an object moves across the image, it will be smeared along its path of movement.

Real-time computer graphics, meanwhile, has traditionally considered a single instant of time for each frame rendered. This is obviously more efficient (and less complicated) than taking into account the appearance of objects over an interval of time. Motion blur can, however, increase the realism of the rendered scene, including smoother animations. It can also be used for artistic or illustrative reasons. Therefore, it has recently started to appear more commonly in real-time computer graphics applications, such as games.

One straightforward way to create motion blur is to use accumulation. The scene is rendered a number of times for different instances of time, and an average result is computed from these. This will converge to correct motion blur as the number of images increases. For a low number of samples, strobing artifacts will appear, however. The problem of this approach is that if the image is rendered $n$ times, the rendering cost will be $n$ times as high. Therefore, more sophisticated techniques have been researched [10]. Most of the techniques that converge toward a correct solution are based on point sampling, such as stochastic rasterization [2]. The drawbacks of these techniques is the presence of noise. In contrast, this thesis concerns motion blur based on analytically computed visibility of geometry [7].

## 1.3 Shadows

Shadow rendering is an important concept in computer graphics because it adds realism and helps the viewer's spatial perception of the scene. A point is considered to be in shadow from a light source if it is occluded from the

light source by another object: computing the shadowing for a point on a surface requires knowledge of possible geometry intersecting the ray between the point and the light source. Therefore, shadows are often generated by the entire scene, or a significant part of the scene. Shadow rendering is a complex problem which has been researched extensively.

One popular technique to render shadows is *shadow mapping*, first introduced by Williams in 1978 [11]. The concept is to view a point (to be shadowed) from the perspective of the light source and perform a depth test (comparing it with other geometry) to determine if it is visible from the light source. The algorithm to achieve this consists of two steps:

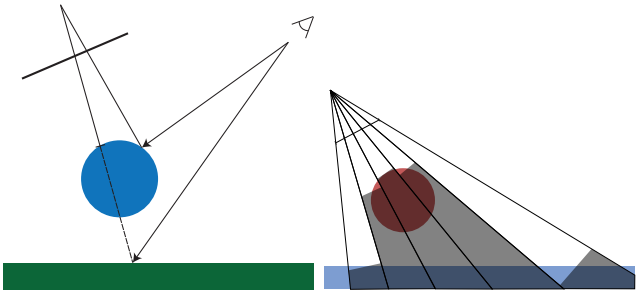
1. The scene is rendered from the point of view of the light source. This perspective will be referred to as the *shadow camera*. The depth values are stored in a *shadow map* so they can be retrieved later.

2. The scene is rendered from the camera's perspective. A linear transformation must first be constructed to transform points from the coordinate system of the camera to the coordinate system of the shadow camera. Each sample point that is rendered is transformed into the light source space, and the depth test between the point and the shadow map is performed, which will determine the visibility of the point from the perspective of the light source. If the point is not visible from the light source, it is in shadow.

Figure 1.2a illustrates how points viewed from the camera are transformed to the space of the shadow map.

Shadow mapping is easy to implement, since only rasterization is needed to compute the depth values. Any object that can be rasterized can be a shadow caster, assuming depth values can be computed for each pixel. This is in contrast to techniques such as *shadow volumes* [5], where the vertices of each object need to be processed. Shadow mapping is also reasonably fast, growing linearly with the number of lights and the size of the scene. And since only the depth values are relevant, shader computations can be disabled during the generation of the shadow map, which can make it faster than the rendering of the scene. Shadow mapping does, however, have some drawbacks.

Shadow maps are generated from the limited perspective of a camera, which cannot properly represent every type of light source. Shadow mapping works best with *spotlights*, since they have a narrow range. The shadow camera frustrum can be computed to cover the range of the light. Other types of light sources cause problems, however. For point lights and directional lights, multiple cameras can be used to cover the entire range of the



(a) Points transformed to shadow map space

(b) Texel coverage

Figure 1.2: Shadow mapping. (a) Shadow mapping for two camera rays. The point on the circle will be lit and the point under the circle shadowed. (b) The distances stored in a shadow map are shown. All points in the transparent grey areas will be shadowed. Aliasing problems are visible: due to the limited resolution of the shadow map, some points will incorrectly be considered to be in shadow if their depths are compared directly to the corresponding depths in the shadow map.

light. Directional lights can also be handled by placing the camera far away enough to cover the entire scene, but this can degrade the shadow quality, as described in the next section.

If there exists multiple light sources in the scene, shadow maps must also be generated for each of these, and the shadow computations for each pixel have to be performed individually with respect to each light source. This thesis assumes the case of a single spotlight, but the techniques described should be straightforward to implement for multiple light sources, as well as point lights and directional lights.

## 1.4 Shadow Quality

Since shadow mapping is an image-based technique, aliasing often appear on shadow receivers. This is caused by *undersampling* of the shadow map, when each texel in the shadow map maps to multiple pixels in the frame buffer. Typical cases when this happens is when the shadow map is far away from the receiver or the resolution is too low, or both. This can be alleviated by increasing the resolution of the shadow map, but this degrades performance. One simple technique to reduce this aliasing is to take the four nearest samples (instead of only one) from the shadow map. Then the depth comparison is performed for each sample, giving four boolean results. These

are bilinearly interpolated to compute the amount of light for the pixel. This can also be done for filter kernels of any size and is called *percentage-closer filtering* [9]. There are many techniques for rendering more realistic soft shadows, which is an important topic in computer graphics that is outside the scope of this thesis.

Incorrect self-shadowing is another common problem with shadow mapping. This is caused by the fact that, most often, a point in camera space will not map perfectly to the center of a texel in the shadow map. All points that map to a certain shadow map texel will be compared to the distance stored in that texel. For a surface that is visible to the shadow camera and not perpendicular to it, there will be different areas where the distances to the light source are, respectively, less than, equal to, and greater than the distance stored in the corresponding texel. This is illustrated in figure 1.2b.

This can be compensated for in different ways. Commonly, a bias is subtracted from the distance. This can however cause missing parts of shadows near shadow casters. Another way is to identify each triangle with an ID and store this in the shadow map. When performing the shadow map comparison, texels with the same ID as the current triangle can be ignored. This will effectively eliminate the possibility that a triangle self-shadows itself. There can still appear incorrect self-shadowing along edges of connected primitives.

## 1.5 Motion Blurred Shadows

If a motion blurred object casts a shadow, naturally the shadow should be motion blurred correspondingly, in order to render a realistic image. This is the topic of this thesis.

### 1.5.1 Previous Work

Lokovic and Veach [8] describe *deep shadow maps*, a technique for high-quality shadows. Instead of storing a single depth per texel, the deep shadow maps store a representation of the fractional visibility at a number of depths. This visibility function takes into account both semi-transparent objects and coverage of pixels.

The authors claim this also can support motion blur, if every shadow image sample is associated with a random time. When the samples are averaged together into visibility functions, this gives average spatial as well as temporal visibility. However, it does not support moving shadow receivers.

Akenine-Möller et al. describe a technique for stochastic motion blur as well as time-dependent textures that can be used as shadow maps [2]. A time-

dependent texture holds multiple time samples in each texel and supports time-dependent reads and writes. The time samples are generated by using random offsets in a grid, such that for $n$ samples per pixel, a time sample $t_i$ belongs to the set $T_i = [\frac{i}{n}, \frac{i+1}{n}]$, for $i \in \{0, ..., n-1\}$. When the shadow map comparison is performed for the screen space sample $t_i \in T_i$, the shadow map sample $t_s \in T_i$ is used. Therefore, $|t_i - t_s| < 1/n$, and the result converges towards the correct image as the number of samples increase. This supports both static and dynamic receivers. Since it is based on sampling, there will be noise.

# Chapter 2

# Semi-Analytical Motion Blurred Shadows

## 2.1 Analytical Visibility

Motion blur rendering can be divided into two parts, *visibility determination* and *shader computations* [10]. This thesis concerns solving visibility with analytical computations. As will be shown, this can be extended with shader computations to perform shadow mapping analytically in the temporal dimension.

Gribel et al. [7] describe a technique for motion blur that analytically computes the visibility of triangles for each pixel, over an interval of time.

For this, *time-continuous triangles* (*TCT*'s) [2] are used. For each TCT, two sets of vertices are specified, one set for *time=0* and one for *time=1*. The vertices are linearly interpolated in time over the interval $[0, 1]$. This means that a vertex $i$ of a triangle at time $t$ is defined as:

$$p_i(t) = (1 - t)q_i + tr_i,$$

where $q_i$ is the vertex at $t = 0$ and $r_i$ is the vertex at $t = 1$.

For each triangle and each spatial sample point, it is shown that a *visibility function*, $v(t)$, can be constructed that equals 1 when the sample point is inside the triangle and 0 when it is outside [7]. By solving $v(t) > 0$, the sub interval of $[0, 1]$ during which the point is inside the triangle can be retrieved.

During this interval, the depth will most often change. This is expressed as the *depth function*. As shown, this function is a cubic rational polynomial. In an implementation, it can however be approximated with a linear function, as the error is rarely significant and it simplifies computations.

*Intervals* are used analogously to fragments in rendering without motion blur. Each interval stores the following parameters:

- the time at the beginning of the interval
- the time at the end of the interval
- the depth function
- the color of the interval

For each pixel in the framebuffer, an *interval list* is stored. Each interval that is rasterized for a pixel is inserted into the list for that pixel. Here, clipping and sorting will be performed, followed by compression in order to reduce the number of intervals. This will save memory and make subsequent operations on the list faster.

After rendering all triangles, a *resolve pass* is performed to compute the final color of all pixels. This is done independently for each pixel by *sweeping* over all the stored intervals. By finding intervals that overlap in time, $[0, 1]$ can be divided into subspans so that for each subspan, one interval will occlude all the others. A *resolved interval* can be created temporarily for each such subspan by clipping the occluding interval to the subspan. If all the resolved intervals are put into a list, the list will be a representation of the original list, but with only visible intervals present. This is illustrated in figure 2.1. The colors of the resolved intervals are integrated against the shutter response function $w(t)$ and added to the final color of the pixel. This can be expressed as [7]:

$$\sum_{k=0}^{n-1}\left(\int_{t_k^s}^{t_k^e} w(t)\boldsymbol{c_k}\,dt\right), \tag{2.1}$$

where $\boldsymbol{c_k}$ is the color of resolved interval $k$ and $t_k^s$ and $t_k^e$ are the start and end times of interval $k$, respectively.

With a box filter, the resolved color is simply the sum of the intervals, weighted by their durations in time.

## 2.2 Shading

The technique described here only computes the visibility. If time is not taken into account in the shaders, only surfaces with a single, constant color will be shaded correctly. Instead, the shading should be integrated over the duration of the rasterized interval.

The color of a pixel can, with some simplifications, be expressed as [10]:

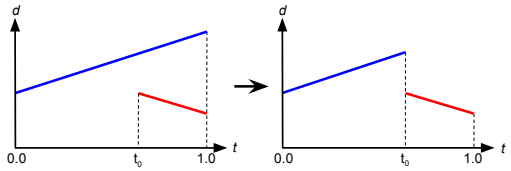



Figure 2.1: Interval list for a pixel. At the beginning of the frame, a blue surface overlaps the pixel, moving away from the camera. At time $t_0$, a red surface sweeps over the pixel at a closer distance, occluding the blue surface, until the end of the frame. The left diagram shows the interval list when both intervals have been inserted. Interval lists can be resolved to remove overlapping parts of intervals, as seen in the right diagram. The final color for this pixel will, with a simple box filter, be a weighted sum of blue and red with proportions $t_0$ and $1 - t_0$ respectively.

$$i(x, y, t) = i(\omega, t) = \sum_l \int_\Omega h(\omega) \int_T f(t) g_l(\omega, t) L_l(\omega, t)\, dt d\omega. \tag{2.2}$$

$\Omega$ is the total solid angle from the environment towards the pixel and $T$ is the exposure time. $h(\omega)$ is the the spatial and $f(t)$ the temporal reconstruction filters. $g_l(\omega, t)$ is the geometry function which equals 1 when object $l$ is visible at time $t$ and solid angle $\omega$, and otherwise equals 0. $L_l(\omega, t)$ is the incoming luminance, or *shading function*.

For each object $l$, the analytical visibility method described here approximates the outer integral by sampling with a box filter. The interval $[t_s, t_e]$ where $g_l = 1$, for each sample, is also computed. Hence, the shading that needs to be computed for each sample is:

$$s(\omega, t) = \int_{t_s}^{t_e} f(t) L_l(\omega, t)\, dt, \tag{2.3}$$

which is the same as equation 2.1 (per object) except that the shading here is time-dependent. With shadow computations, $L_l(\omega, t) = f_s(\omega, t) L_{ls}(\omega, t)$, where $f_s(\omega, t)$ is the fraction of shadow and $L_{ls}(\omega, t)$ is the result of other shading computations. A more general solution to time-dependent shading is outside the scope of this thesis and it is instead assumed that $L_{ls}(\omega, t) = \boldsymbol{c_l}$, which can be factored out of the integral.

A problem is that during rasterization for each polygon, only the interval when the polygon overlaps each sample point can be computed. Occlusions

by geometry can only be determined if the occluding polygons are already rasterized for the pixel, as their depths are stored in interval lists similarly to standard z-buffering. This is sufficient when polygons are rendered in back-to-front order, but no such assumption is made here. Therefore, time-dependent shading for a pixel should ideally be deferred until all geometry have been rasterized for that pixel. It will, however, not be very noticeable in most cases.

## 2.3 Shadows

When objects move relative to a light source, the shadows cast on other surfaces in the scene should be motion blurred. This is because points on surfaces in the scene may be occluded by the light source for only a part of the frame.

This means that to achieve correct motion blurred shadows with shadow mapping, a time-dependent shadow map is needed. For the shadows to be analytically motion blurred, the shadow map needs to store analytical visibility data from the perspective of the light source.

### 2.3.1 Static Receiver

A *static receiver* is a sample interval with a constant depth function. This most commonly appears with objects that are static with respect to the camera, but also in other cases, such as surfaces moving sideways. Since the depth function is constant, the interval can be represented by a single point, which maps to one point in the shadow map. The shadowing for the interval is the fraction of time the point is in shadow during this interval.

The temporal resolve method described in section 2.1 can also be used to compute the shadowing for a point. For a texel in the shadow map, the interval list holds the depth of possible occluders during $[0, 1]$. An interval $i_d$ with the duration and depth (with respect to the shadow camera) of the sample interval can be inserted into the interval list. The point is in light when it is visible (not occluded) from the light source, i.e. during the sub intervals where the depth of $i_d$ is less than all the other intervals.

If the colors of all original intervals are set to black and $i_d$ is set to white, the resolved interval list will have white intervals where the point is in light and black where the point is in shadow. This is illustrated in figure 2.2. The list can be resolved according to equation 2.1, and divided by the length of the sample interval, to compute the fraction of light.

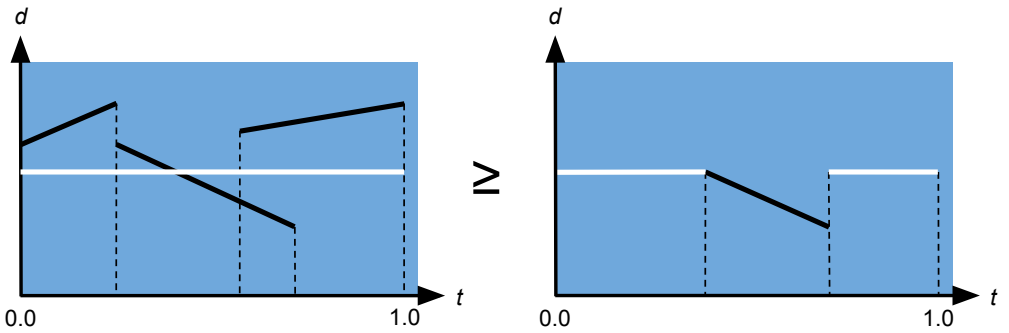



Figure 2.2: The interval list for a texel in the shadow map (black intervals) with the added interval representing the depth of a point in the scene (white). To the left are the (uncompressed) intervals, to the right the resolved intervals. The color, which here represents the fraction of light for the pixel, is the sum of the resolved intervals, weighted by their respective durations in time.

### 2.3.2 Dynamic Receiver

For a dynamic shadow receiver, the depth function varies with time. If the end points of the interval map to the same point in the shadow map, the shadowing can be computed with the same method as a static receiver. The interval that is inserted into the list of the texel should then have depths at the end points corresponding to the end points of the sample interval. If the interval maps to different but very close points in the shadow map, the same method could be used, as an approximation. These cases can appear when the camera and shadow camera are closely aligned or the change in depth is very small. The remaining part of this section assumes cases where this does not apply.

When the end points of the interval map to two points in the shadow map, a line is formed which can intersect multiple texels. For each instant of time during the interval, the shadowing is determined by the time and the current position according to the depth function. This means that for correct shadowing, all texels the interval sweeps over in the shadow map contributes to the result. An example of this is shown in figure 2.3.

The shadowing for a spatio-temporal interval defined by the function $p(t) \in \mathbb{R}^3, t \in [t_s, t_e]$ can be expressed, in accordance with equation 2.3, as:

$$\int_{t^s}^{t^e} f(t)s(d_p(p(t)), d_s(p(t), t))\, dt, \tag{2.4}$$

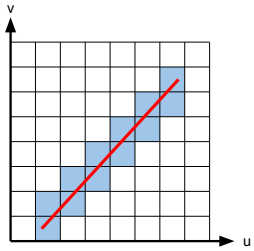



Figure 2.3: A sample interval transformed into shadow map space, sweeping over several texels.

where $f(t)$ is the filter and $s(d_a, d_b)$ is the depth comparison (which returns 0 or 1). $d_p(p)$ is the depth of $p$ as seen from the light source and $d_s(p, t)$ is the distance to the nearest occluder at time $t$ intersecting the ray from $p$ to the light source, and $p(t)$ is the point at time $t$. In comparison, for a static receiver, $p(t)$ would not be time-dependent. Furthermore, $p(t)$ can map to different texels for different sub intervals of $[t_s, t_e]$.

### Backfacing Polygons

In 3D graphics, polygons are often considered to be one-sided. When rendering, polygons which are facing away from the camera can be discarded from rasterization. This can be determined by computing the dot product between the polygon normal and the direction vector from the polygon to the camera (the *view direction*). If the dot product is less than zero, the polygon is backfacing.

This is also useful for shadow rendering. Surfaces that are facing away from the light source must be in shadow, and no further processing is necessary. When computing the dot product for this, the view direction is substituted with the *light direction*, which is the direction vector from the polygon to the light source.

However, when polygons move relative to the light source, they can turn away from, or towards, the light source during a sample interval. Both the normal and the light directions are interpolated in time. With a linear interpolation of the vector elements, the time dependent dot product between them can be written as:

$$f(t) = (n_0(1-t) + n_1 t) \cdot (d_0(1-t) + d_1 t), \tag{2.5}$$

where $n_0$ and $d_0$ are the normal and light direction at the start of the interval, respectively, and $n_1$ and $d_1$ are the corresponding vectors at the end of the interval. The polygon is then visible from the light source when

(a) (b) (c)

Figure 2.4: A surface turning away from the light source during a rendering interval. In (a), it is completely lit, in (c) completely dark and in (b) in between.

$$f(t) > 0, \quad 0 \leq t \leq 1. \tag{2.6}$$

This is a quadratic equation which can be solved analytically to find 0, 1 or 2 intervals during which the polygon is oriented towards the light source. These intervals are in proportion to the interval being rasterized. The remaining shadow calculations then only need to be performed for the visible intervals, and the result from each one is weighted in proportion to the interval's length.

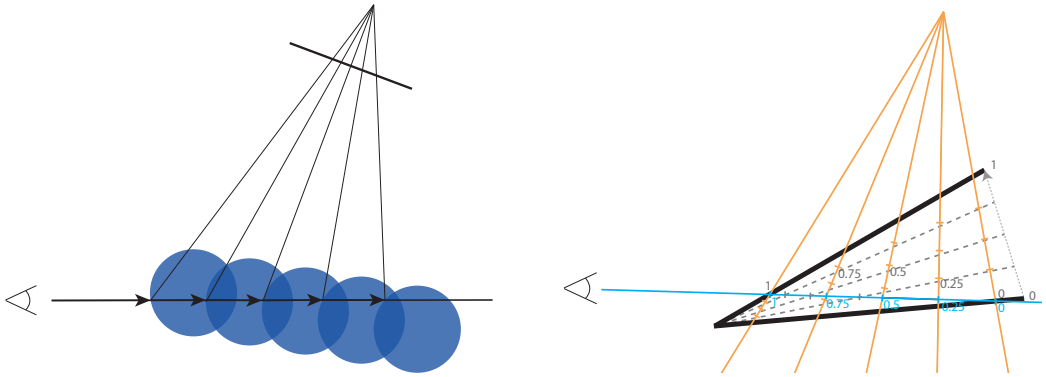
In figure 2.4, three successive frames from the implementation are shown. The plane is initially facing the light source, but turns away from it during the second frame. The transition from light to dark can be seen in figure 2.4b.

### Point Sampling

The function in equation 2.4 can be approximated by point sampling. The sample interval is divided in time into a number of sample points. For each sample time $t_i$, the position of the point is computed by interpolation, and the corresponding texel in the shadow map is retrieved. The shadow map comparison is then performed for the fixed time $t_i$ and the depth of the sample point, by finding the nearest occluder at $t_i$. This can also be extended to spatial filtering, such as percentage-closer filtering, for each sample point.

Sampling of a dynamic receiver is illustrated in figure 2.5a. A sphere moving away from the camera results in a decreasing depth function, which is divided into five sample points.

The number of samples could be computed as proportional to the length of the interval in shadow map space. This will take into account the movement of the receiver. An interval that sweeps over a large number of texels will




(a) (b)

Figure 2.5: Dynamic receivers. (a): A sphere moving away from the camera, sampled five times. (b): A rotating surface viewed from a shallow angle. The sample interval is divided into five sample points, marked with the corresponding times in the figure. The correct interpolation of the surface (and the sample points) are drawn with dashed lines. The depths interpolated in the shadow map are incorrect as well, though not as severely. (The resolution of the shadow map is ignored in this example.)

overlap each individual texel for a very short time interval, and a point sample for each is therefore a reasonable approximation. For shorter sweeps in the shadow map, this will give worse results. Solutions to this is either to set a minimum number of samples, or to treat each sample point as a static receiver when there is a low number of samples.

A more analytical solution to this is outside the scope of this thesis. See chapter 5 concerning dynamic receivers.

Problems can appear when non-linear depth functions are approximated as linear functions. The end points of all intervals will have the correct depths, but this is not the case for sample points generated by linear interpolation between the endpoints. This first affects the sampling of the sample interval. The interpolated depth will not be the correct depth for the sample time, and the position may also project to the wrong texel in the shadow map because of this. When the depth test in the shadow map is performed, both the depth of the sample point and the depth at the time in the interval list of the texel will be incorrect. An example of this, with a rotating surface resulting in a non-linear depth function, is shown in figure 2.5b.

To alleviate these problems, the depth function can be split into a number of sub intervals by sampling the depth at different times during the interval. This list of intervals will then be a closer approximation of the depth function than just one interval, as illustrated in figure 2.6. However, this will require

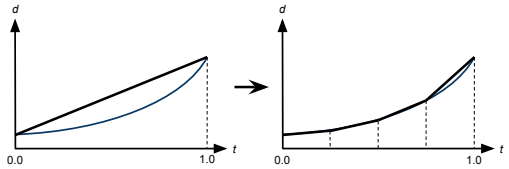



Figure 2.6: Depth sampling. To the left the depth function is sampled only at the endpoints of the interval. To the right, it is split into four linear intervals, to minimize the error.

sampling of each sample interval, and if stored in the shadow map, more space. This could also affect interval list operations, such as when shadow map intervals are inserted. Additionally, in many cases the linear function is very close to the actual depth function which would make extra depth sampling unnecessary. This could perhaps be handled with adaptive sampling. See section 5.

Incorrect self-shadowing is an inherent problem of shadow mapping, and there are a variety of techniques to handle it, as described in section 1.4. With motion blurred shadows, these problems are present in addition to the problems specific to motion blur.

As in other cases of incorrect self-shadowing, this can be compensated for by introducing a bias. If the maximum error of the sample interval and the intervals in the texel are known, these can be used as added biases. This will compensate for the incorrect depth, but not if the wrong texel is retrieved. However, there are disadvantages to this. Increased bias can as always cause parts of shadows to disappear when the shadow caster is close to the receiver. And the maximum error will need to be computed, either analytically or by sampling. If it is used in the shadow map, it also needs to be stored, for all intervals in each texel.

Another method to solve the problem of self shadowing in general is to uniquely identify all polygons, as described in section 1.4. The ID's are written to the shadow map for each interval of each pixel. When the depth test is performed, intervals with the same ID's as the current polygon can be discarded.

Polygon identification is not a complete solution, however. Incorrect self shadowing could still appear along the edges of connected polygons, e.g. along the diagonal of a square made up by two triangles. Geometric computations could be performed to detect groups of polygons that cannot self-shadow each other (such as polygons that lie in the same plane), and give them the same ID. But there would still remain many instances of connected polygons self-shadowing each other.

# Chapter 3

# Implementation

The implementation builds on the rendering framework described by Gribel et al. [7], where the core is a software rasterizer supporting analytical motion blur. It can be roughly divided into three parts: application, renderer and shaders. As usual in real-time graphics applications, the application is responsible for configuring the renderer and creating the scene, as well as handling user input and performing animations etc. One important task is to set input data to the shaders, such as matrices. The pixel shaders emulate normal programmable pixel shaders, and use the input data (global as well as per-vertex) to compute the color for each pixel.

The application and shaders of a graphics application usually have to cooperate to support shadow mapping; the application provides the input and the shaders perform the shadow mapping computations per-pixel. To enable motion blurred shadows, the rasterizer also had to be modified to compute and provide analytical visibility data as input to the pixel shaders.

## 3.1 Rasterizer

The two substantial extensions that were necessary for the analytical rasterizer to support shadow mapping was support for external depth targets, and sending interval data to the pixel shader.

The external depth target is needed when the shadow map is generated. The application creates a depth buffer and informs the rasterizer that it should render depth values to it. Multiple threaded rasterizers are used, each one saving the depth values in their own tile. The global color buffer that is written to the screen only contains the resolved values. Therefore, an external depth target that is written to by each rasterizer is needed. When it is enabled (set by the application), the intervals that are written to each

tile are also written to the depth target.

When the shadow map is generated, only the depth values are of importance. Therefore, executing pixel shaders and writing colors can be disabled in order to save some rendering cost.

Interval data is sent to the pixel shader in a structure. It contains the interval endpoints and the start and end positions (in clip space) of the pixel that is being rendered. The position of a pixel at a certain time is computed by interpolating the vertex positions with perspective-correct time-dependent barycentric coordinates.

## 3.2 Shader

The algorithm that computes the shadowing for analytically motion blurred objects is implemented as a shader procedure. It accepts the following arguments:

- the interval endpoints in time,
- the start and end positions of the interval,
- the shadow map,
- normals for *time=0* and *time=1* in the view space of the shadow camera,
- a matrix to transform from the clip space of the camera to the clip space of the shadow camera,
- and a matrix to transform from the clip space of the camera to the view space of the shadow camera

The interval data is sent by the analytical rasterizer as input to the pixel shader, as described in section 3.1. The other parameters must be provided by the application.

Most importantly, these parameters are used to calculate the start and end position of the interval in the clip space of the shadow camera. This is needed for the shadow map lookup. The normals as well as direction vectors from the point to the light source, for the endpoints of the interval, are also obtained.

The sub intervals where the surface is facing the light source are computed with the method described in section 2.3.2, using the normals and light direction vectors.

The shadowing is computed individually for each sub interval facing the light source. The start and endpoints of the sub interval (as seen from the shadow camera) now constitute the endpoints of a line through the shadow map. If the endpoints map to the same texel in the shadow map, the analytical occlusion procedure for the texel, as described in section 2.3.1, is used.

In the depth test function, the interval list for the texel is first copied from the shadow map. If intervals with the same polygon ID as the current rasterized interval exist in the list, these are removed. Then the interval representing the depth of the point (colored white) is inserted and the color of the interval list is resolved and divided by the length to compute the fraction of light. It is assumed that all intervals in the shadow map are colored black, otherwise the color could be set in this procedure.

If the line sweeps through many pixels, the receiver is dynamic. A number of samples proportional to the length of the interval in shadow map space is computed, and point sampling over the interval is performed. A limit for a minimum number of samples is used, in order to achieve good quality for slow-moving objects. The shadow map interval list for each sample point is retrieved and intervals that overlap the sample point in time are found. If any of these has a depth (at the time of the sample) that is less than the sample point, the point is in shadow and otherwise not.

Bilinear filtering was implemented and can be optionally enabled. Instead of one depth test (analytical occlusion computation or point sampling), four depth tests are made for the neighboring texels and the results are bilinearly interpolated.

## 3.3 Application

The application creates the objects in the scene for motion blur in the usual way. An ID is generated for each polygon that is created. It also creates a shadow map and renders two passes: one to generate the shadow map, and one to render the scene. In the first pass, the shadow map is set as the depth render target for the renderer. In the second, it is instead set in the render states of the objects so the pixel shader can access it.

Similar to normal shadow mapping, the application creates matrices for the shaders. Two normal matrices are computed and input to the shader, for $time = 0$ and $time = 1$, respectively.

Before rendering each frame, animation is performed by setting the matrices for $time = 0$ and $time = 1$, respectively, for the objects.

# Chapter 4

# Results

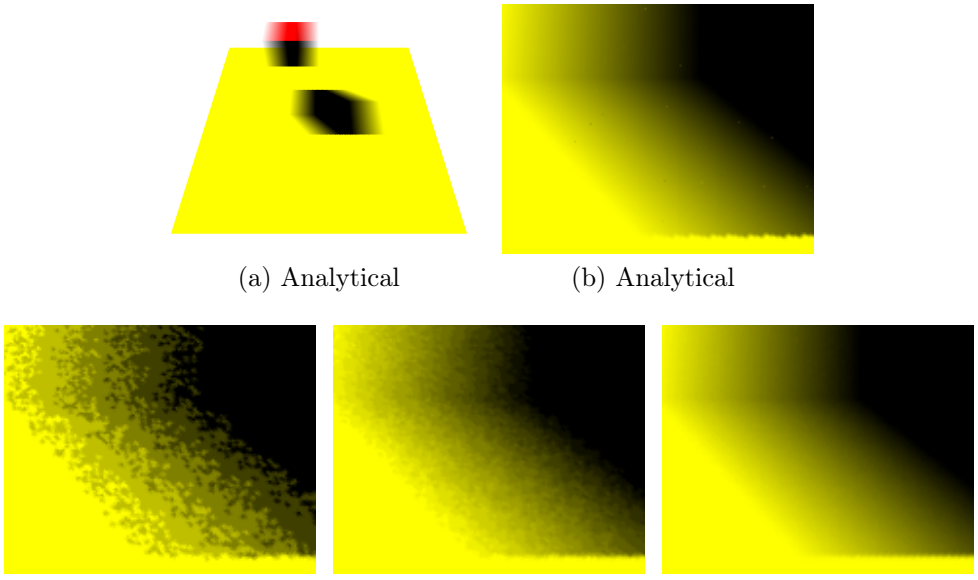
## 4.1 Image Quality

The motion blurred shadows on static receivers in the generated images appear to have a high quality, whereas the shadows on dynamic receivers contain artifacts in some cases, for example when the angle between the surface and the direction to the light source is small.

Figure 4.1, 4.2 and 4.3 show a very simple scene, consisting of a cube and a plane. In the first figure, the plane is static and in the second and third, dynamic. Analytical and stochastic rasterization is compared, and the absence of noise in the shadows with analytical motion blur is noticeable for both static and dynamic receivers. The number of samples used for dynamic receivers when rendering these images were computed as $max(2|p_e - p_s|, 5)$, where $p_s, p_e \in \mathbb{R}^2$ are the start and end points of the interval in shadow map space. Percentage-closer filtering was used for both the analytical and stochastic shadows in both figures. When the plane is rotating in this scene, some inconsistencies in the shadow can appear near the rotational axis, because there is a transition between static and dynamic receiver. This is shown in figure 4.3a, and can be alleviated by increasing the number of samples. For this simple scene, incorrect self-shadowing was efficiently eliminated by using polygon ID's.

The implementation does not use spatial antialiasing, which, on the contrary, is included in stochastic rasterization. A jagged edge resulting from this is visible in figure 4.1b. For the dynamic receiver in figure 4.3, the area of the shadow corresponding to this edge contains artifacts as well. In this case, increasing the number of samples does not improve the result, as shown in figure 4.3e. Analyzing this more closely has been left for future work.

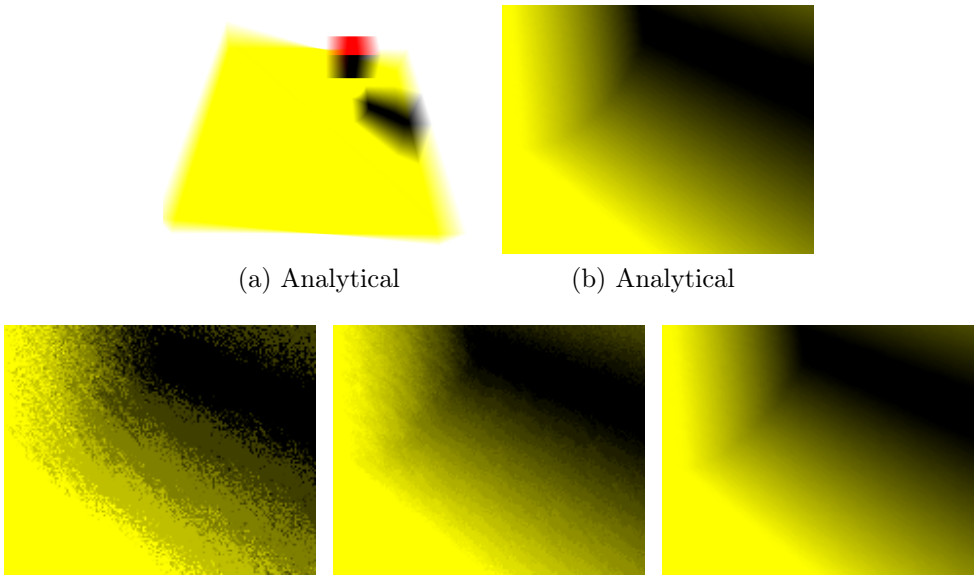
Figure 4.4 and 4.5 show a highly tessellated scene, consisting of nearly



(a) Analytical (b) Analytical

(c) 4 samples (d) 16 samples (e) 256 samples

Figure 4.1: A motion blurred cube casts a shadow on a plane. Figure (a) and (b) show an analytically computed shadow with 8 intervals. The remaining images show the scene rendered with stochastic motion blur as reference.



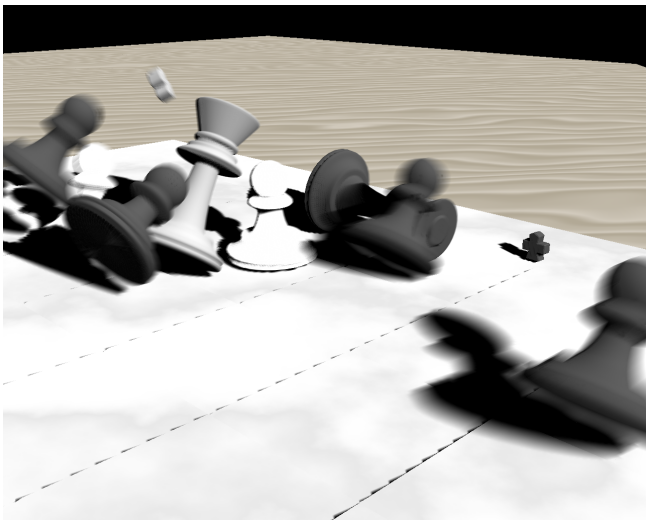
(a) Analytical (b) Analytical

(c) 4 samples (d) 16 samples (e) 256 samples

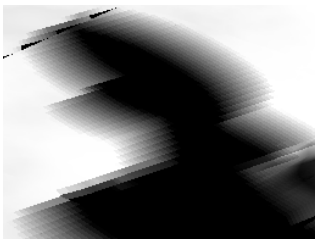
Figure 4.2: The same scene as in figure 4.1, but with a rotating plane.

(a) Dynamic (b) Dynamic (c) 256 samples (d) Dynamic (e) 256 samples

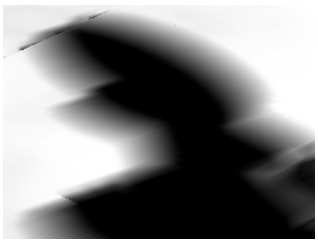
Figure 4.3: The same scene as in figure 4.2, but with less movement of the plane and a differently positioned cube. (b) and (c) show a zoom-in of the area of the shadow close to the rotational axis of the plane. (c) and (d) show a zoom-in of an area with different artifacts that are not caused by undersampling. The images in (a), (b) and (d) are computed dynamically per rasterized interval, as described in section 4.1. (c) and (e) show reference images generated by using 256 samples per interval.



(a) Percentage-closer filtering

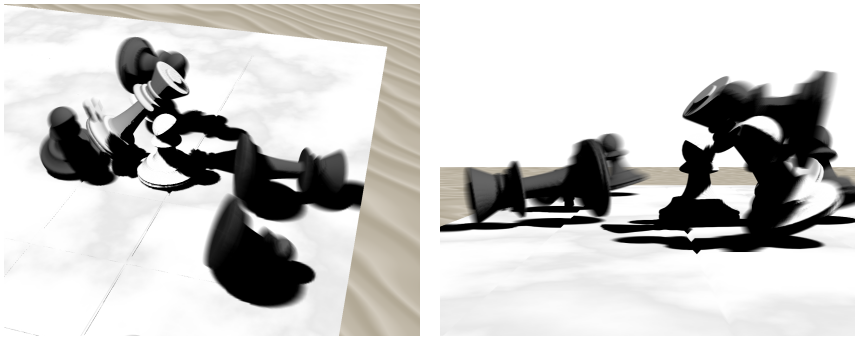


(b) No filtering



(c) Percentage-closer filtering

Figure 4.4: A highly detailed scene rendered with shadows. (b) and (c) show a zoom-in of a static receiver, illustrating the improved quality when percentage-closer filtering is used.



(a) (b)

Figure 4.5: Different views from the scene in figure 4.4, at a slightly different stage in the animation.

30000 triangles. Since the polygons are very small, most of the rasterized intervals for the moving objects have short enough durations to be handled as static receivers. The high triangle density also causes a large number of cases of incorrect self-shadowing that cannot be handled by identifying polygons with ID's. Therefore, a higher bias value was used for this scene. Figure 4.4 also shows a comparison between percentage-closer filtering and no filtering for a static receiver.

The quality of the rendered images could be improved further by adjusting shadow mapping parameters such as the perspective of the light source and the resolution of the shadow map. This can increase the detail and reduce aliasing of the shadows, and is as relevant for static as motion blurred shadows.

## 4.2 Performance

The implementation was tested on a system with a 2.4 Ghz Intel Core 2 Duo CPU. Rendering was performed at a resolution of $1280 \times 1024$ pixels, using a single thread for the rasterizer.

The rendering time for the image in figure 4.1 was about 2.5 seconds, of which the shadow computations took about one second. For the image in 4.2, it was about 18 seconds, with 16 seconds to compute the shadows. When the number of samples for dynamic receivers was lowered to $max(|p_e - p_s|, 5)$, the shadow computation time decreased to about eight seconds. Since the number of samples are proportional to the length of the interval in shadow map space, the computation time is dependent on the angles of the camera and the shadow camera, as well as the angle and motion of the dynamic

receiver.

The scene with the chess board also varies significantly in terms of rendering time, depending on many different factors. Times up to 40 seconds for a frame have been observed, when the camera is close to the highly detailed chess pieces.

Performance was not the focus of this thesis, and further investigations into testing and analyzing rendering times and optimizations and has therefore been left for future work.

# Chapter 5

# Conclusions and Future Work

It has been shown that analytically motion blurred shadows are possible to implement and are an important component in rendering high-quality scenes with motion blur. Much work remains to be done however, including optimizations, reducing errors and improving quality with techniques such as antialiasing and soft shadows. Furthermore, shadow mapping is a technique with parameters that must be finely adjusted in order to achieve good quality and avoid artifacts. This could be investigated more thoroughly for scenes with motion blurred shadows; there can be errors inherent to shadow mapping, errors due to inaccuracies of analytical visibility (such as compression or the approximation of the depth function), and errors due to the combination of both.

### Shading

Combining shadow mapping with other time-dependent shading functions (such as lighting or texturing) for motion blur is important for rendering high-quality scenes. To solve this correctly, the shadowing and the supplementary shading function should be integrated together, as suggested by equation 2.3. In this case, the shadowing could, for example, be computed as an interval list which could be multiplied with time-dependent results of other shading functions.

### Analytical Dynamic Receivers

Instead of point sampling for dynamic receivers, an analytical technique could be used in order to compute more accurate shadows. A traversal algorithm should be applied to the interval in the shadow map to determine exactly which texels it intersects and the durations of these intersections. This should

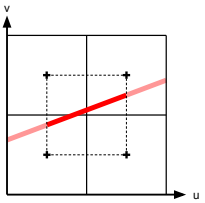



Figure 5.1: The sample interval in shadow map space, sweeping over a four-texel filter region. This could be computed analytically.

work in the same way as texturing a motion blurred surface. For each texel, the one-texel analytical occlusion procedure could be applied with the corresponding sub interval. Filtering this is more complicated, since the proportions used for the filtering are time-dependent. See figure 5.1.

### Adaptive Depth Sampling

To minimize errors due to incorrect depth interpolation, adaptive depth sampling could be used. The depth function would be split into multiple intervals for a better approximation as described in section 2.3.2 but only where there is a significant difference between the real depth function and the linear approximation. As shown in figure 2.6, parts of the interval could be adequately approximated by a linear function, and other parts not. The sub intervals would be processed recursively. This would have the benefit of multisampled depth, but without decreasing performance much.

A problem with this approach is that the intervals would have variable size, which would make storage in a depth buffer complicated. Instead, these multisampled intervals could be split into different intervals when inserted into interval lists.

One solution could be to use multisampled depth functions as input to pixel shaders (since this is probably more efficient than executing the pixel shader multiple times for the sub intervals), but store split intervals in buffers.

### Variance and Exponential Shadow Maps

Motion blurred shadows suffer from aliasing in the same way as standard shadows maps. As shown, PCF is possible to implement, but a major drawback is that the filtering cannot take place until after the depth tests are performed. Variance [6] and Exponential [3] shadow maps are techniques that address this problem, efficiently enabling the use of pre-filtering such as mipmapping [12]. This could be very beneficial to motion blurred shadows as well.

### Transparency

Transparent objects can cast transparent shadows, and these should be motion blurred as well. This obviously makes computations more complicated and has therefore been omitted from this thesis.

# Bibliography


[1] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering, 2nd edition*. A. K. Peters, Ltd., 2002.

[2] Tomas Akenine-Möller, Jacob Munkberg, and Jon Hasselgren. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, pages 7–16, 2007.

[3] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential Shadow Maps. In *Graphics Interface*, pages 155–161, 2008.

[4] David Blythe. The Direct3D 10 System. In *ACM SIGGRAPH 2006 Papers*, pages 724–734, 2006.

[5] Franklin C. Crow. Shadow Algorithms for Computer Graphics. In *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 242–248, 1977.

[6] William Donnelly and Andrew Lauritzen. Variance Shadow Maps. In *Symposium on Interactive 3D Graphics and Games*, pages 161–165, 2006.

[7] Carl Johan Gribel, Michael Doggett, and Tomas Akenine-Möller. Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics*, pages 163–172, 2010.

[8] Tom Lokovic and Eric Veach. Deep Shadow Maps. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH, pages 385–392, 2000.

[9] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering Antialiased Shadows with Depth Maps. *SIGGRAPH Computer Graphics*, 21:283–291, August 1987.

[10] Kelvin Sung, Andrew Pearce, and Changyaw Wang. Spatial-Temporal Antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):144–153, 2002.

[11] Lance Williams. Casting Curved Shadows on Curved Surfaces. *SIGGRAPH Computer Graphics*, 12:270–274, August 1978.

[12] Lance Williams. Pyramidal Parametrics. *SIGGRAPH Computer Graphics*, 17:1–11, July 1983.