Point Sampling Algorithms for Simulating Motion Blur

Ryan Meredith-Jones

December 18, 1999

Contents

List of Figures

Chapter 1

Introduction

In computer graphics, *rendering* refers the the process of converting a digital description of a particular scene into a digital image of the scene as observed from some viewpoint. The holy grail of rendering is *photorealism*, that is, the production of synthetic images that are indistinguishable from photographs taken by a real camera. In the pursuit of photorealism, the vast majority of rendering research has concentrated on simulating the appearance of various objects under different lighting conditions. In real photography, however, the appearance of the final image depends just as much on the properties of the camera taking the photograph as it does on the scene being photographed. In order to achieve photorealistic results, rendering software must simulate visual effects resulting from the the optical properties of the camera as well as from those of the scene.

One important camera effect that must be simulated is *motion blur*, in which a moving object appears blurred along its trajectory in a photograph. This effect is the result of the camera aperture being held open for a short amount of time (called the *exposure time*) when a photograph is taken, with longer exposure times and faster object motion resulting in more pronounced blurring. Figure 1.1 illustrates motion blur using several actual photographs of a person's hand. In the first picture, the hand is not moving, so no blur is apparent. The pronounced blurring in the second photograph was produced by shaking the hand rapidly up and down during the exposure time.

Although motion blur is useful for conveying a sense of movement in a single synthetically generated image, it becomes essential when generating a sequence of successive frames for an animation. In such an animation, human viewers expect moving objects to exhibit motion blur. If a synthetically rendered moving object fails to blur appropriately over an animation, the illusion of reality is destroyed in the minds of the viewers. Thus, motion blur is an important phenomenon for rendering software to simulate.

Stationary Hand **Moving Hand**

Figure 1.1: Motion blur in an actual photograph of a human hand.

1.1 Motivation

Rendering is widely regarded as the most computationally intensive step in the process of creating of computer generated imagery. Most rendering algorithms only generate "instantaneous" (i.e. non-motion-blurred) images, by computing the appearance of a scene (as seen through some virtual camera) at one particular instant in time. By contrast, simulation of motion blur requires the renderer to consider changes in the appearance of the scene (due to moving objects, changing lighting conditions, etc) during the exposure time. In a real camera, the film accumulates light energy from the scene while the shutter is open, producing a final image that is essentially a combination of the appearance of the scene at every time in the exposure time interval. This idea is illustrated in Figure 1.2.

One approach to simulating this process in a renderer is to sample the intensity of the light energy incident upon various points on the film surface (known as the *image plane* in graphics) at various times over the exposure time interval. By blending together the sampled intensities, an approximation of the true motion-blurred image can be constructed. As the number of point samples is increased, the approximation eventually becomes indistinguishable from the exact result. Figure 1.3 shows several approximate images computed using this technique, known as *point sampling*.

Each point sample of the intensity function is expensive to compute, so it is desirable for a point-sampling motion blur algorithm to converge to an acceptable approximation using as few samples as possible. The quality of this approximation is highly dependent on the *sampling strategy* that is used to select the times at which the intensity function is to be sampled. In this thesis, we present several new sampling strategies for motion blur, with the goal of producing pleasing motion-blurred images using fewer point samples than

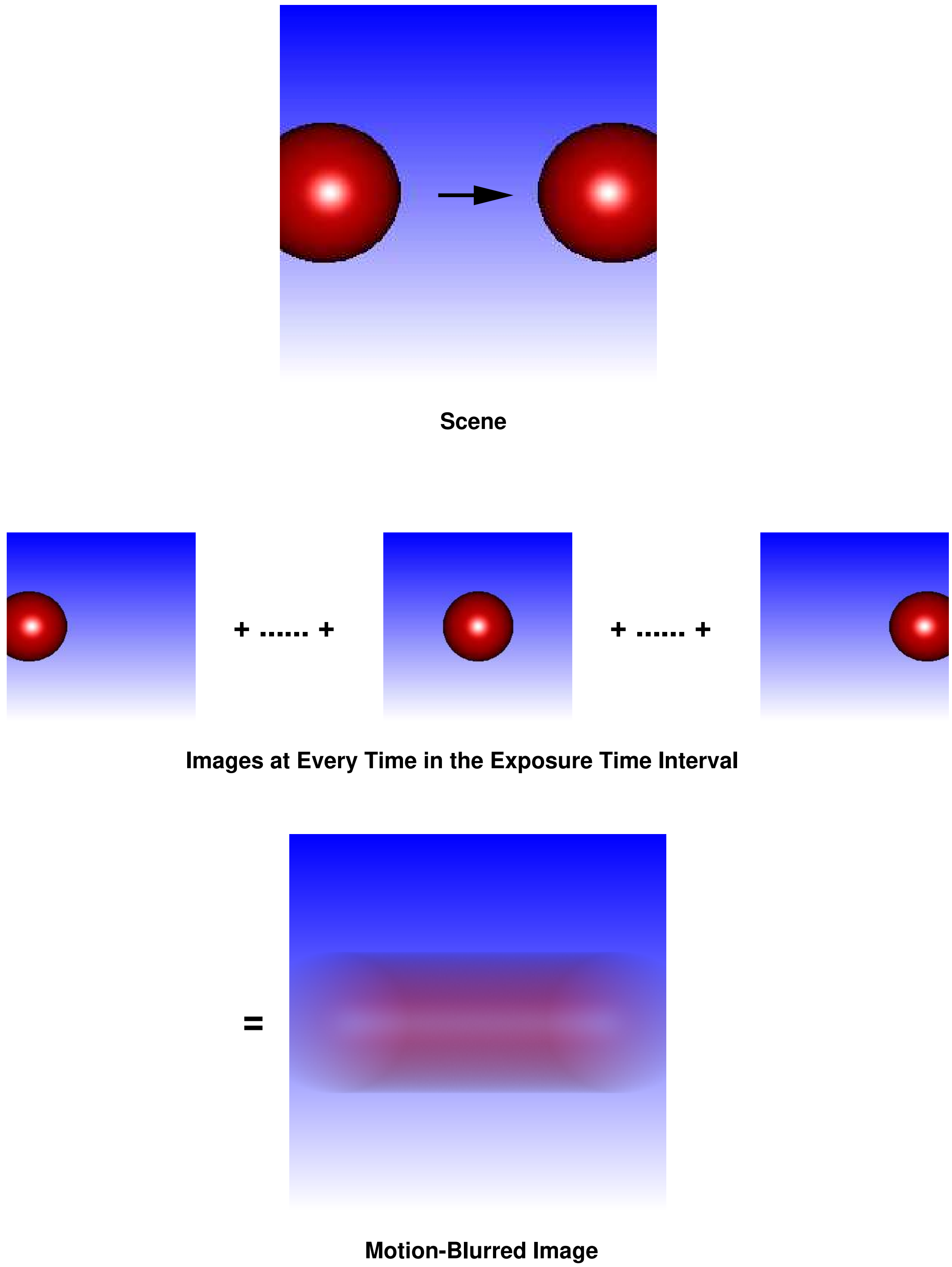
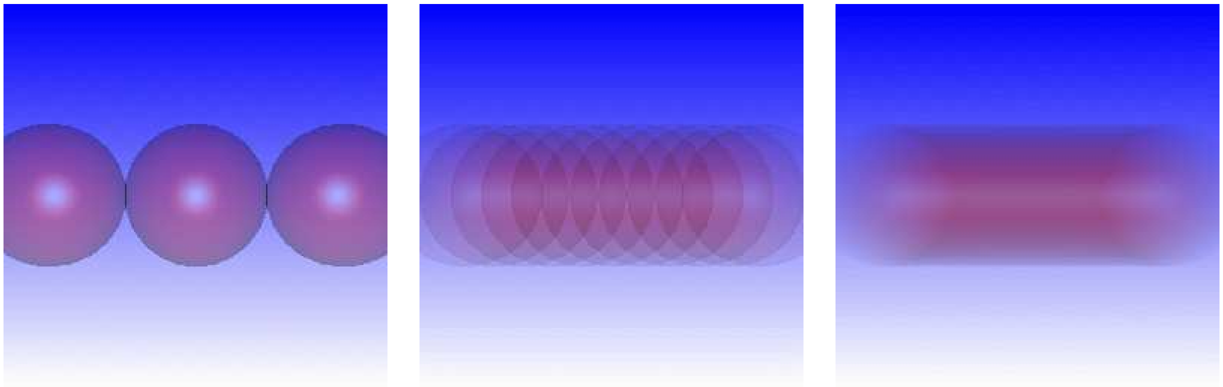


Figure 1.2: Formation of a motion-blurred image. In a real camera, a motion-blurred image is essentially a combination of pictures of the scene taken at every single time in the exposure time interval. In the scene above, a sphere moves across the camera's field of view. We show several of the infinite number of instantaneous images of the sphere during the exposure time, which combine to form the final motion-blurred image.



3 Point Samples/Pixel **10 Point Samples/Pixel** **100 Point Samples/Pixel**

Figure 1.3: Motion-blurred images produced using point-sampling. The point samples are uniformly spread over the exposure time interval, and the same sampling pattern is used at each pixel. As the number of samples is increased, the point-sampled images approach the true motion-blurred image (shown in Figure 1.2).

existing strategies.

1.2 Scope

When approximating a function using point sampling, the quality of the approximation is influenced by the *variability* of the function, or how fast it varies over time. The more variation exhibited by the function between the sample times, the more information about the function that can be missed by our sampling strategy, resulting in a poor approximation. Figure 1.4 shows a highly variable function, and a set of point samples that fail to capture several important features of the function. In this thesis, we examine the causes of variation in a scene's intensity function over time, and present sampling strategies specifically tailored to cope with this variation.

1.2.1 Visibility

One source of variability in the intensity function is due to changes in *visibility* over time. As objects move during the exposure time interval, the object visible at a particular point on the image plane may change, resulting in an instantaneous change (or *discontinuity*) in the intensity function for that point. Existing sampling strategies for motion blur do not cope well with these discontinuities, resulting in unpleasant banding artifacts in the resulting images. These banding artifacts are apparent in the 10-sample image in Figure 1.3. Instead of falling off smoothly towards the left and right edges of the moving object, the colour falls off in abrupt steps, resulting in the distinct colour bands in the image. These discontinuities in the

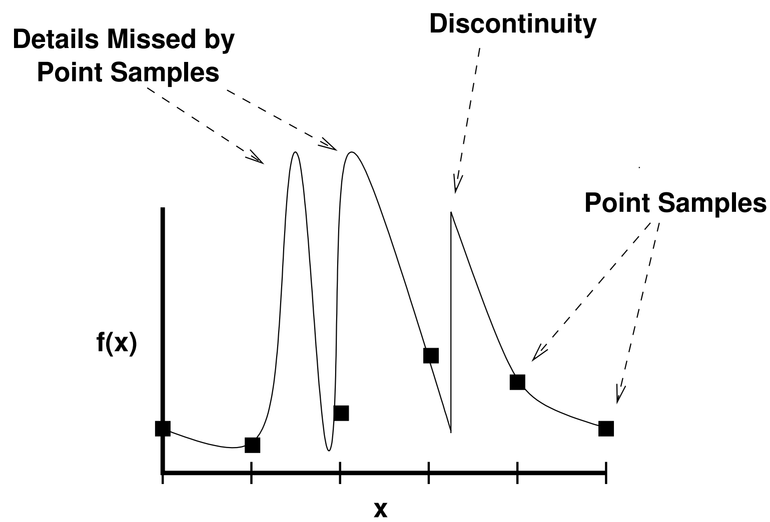


Figure 1.4: Point-sampling a function. If the point samples are spread too far apart, they may fail to capture the variation of the function. In the example shown, the point samples fail to notice the two peaks in the function, since the peaks occur between point samples. The samples are also too far apart to accurately estimate the location of the discontinuity in the function.

image intensity are the direct result of the discontinuities in the underlying intensity function that is being sampled.

In Chapters 3 and 4, we discuss new sampling strategies that eliminate these artifacts by actively seeking out the locations of the "visibility discontinuities" in the intensity function. Chapter 3 is based on work by Ouellette and Fiume [11], and presents sampling strategies for locating up to 2 visibility discontinuities per pixel, which is adequate for handling simple scenes involving a single object. In order to handle more complicated scenes, these ideas must be extended to handle an arbitrary number of visibility discontinuities, which we do in Chapter 4.

1.2.2 Shading

The other main source of variation in the intensity function is due to changes in the observed colour of a particular moving object over time. Typically, lighting conditions and material properties will cause the colour of an object to vary over its surface. As an object moves, a particular point on the image plane receives light from different parts of the object, therefore the observed intensity function at that image plane position will vary over time. This *shading variation* may be slight, if the object has a relatively constant colour, or severe, if the object's colour varies quickly over its surface. In Chapter 5, we present sampling strategies that attempt to capture this variation for several different classes of surface material. We also discuss how to combine these strategies with our visibility sampling strategies, with the result being a complete sampling strategy for computing motion-blurred images.

1.3 Overview

In Chapter 2, we formulate the motion blur problem in mathematical terms. Using this formulation, we examine the merits of several previously developed motion blur algorithms. Chapters 3 and 4 describe our visibility sampling strategies. Chapter 5 describes our shading sampling strategies, and how to integrate them with our visibility results. In Chapter 6, we present our conclusions, as well as ideas for future study based on this work.

Chapter 2

Previous Work

2.1 Introduction

The problem of motion–blurred image synthesis has been studied by graphics researchers since 1983, and a number of different algorithms have been proposed. In this chapter, we attempt to categorize and describe these various algorithms using a consistent mathematical framework. After formulating the problem in mathematical terms in Section 2.2, we discuss analytic algorithms for simulating motion blur in section 2.3. Algorithms based on Monte Carlo integration are covered in section 2.4, while postprocess algorithms are dealt with in section 2.5. Finally, we conclude with a discussion of motion blur algorithms based on geometric substitution in section 2.7.

2.2 Problem Formulation

We shall formulate the motion blur problem in the context of a highly simplified projective model of the image formation process. The virtual environment is expressed in terms of a viewing coordinate system, centred at the origin O. Light from the environment is projected onto an image plane perpendicular to the z-axis. The amount of light impinging on the image plane is quantified by the intensity function $I(x, y, t)$, which measures the incoming light intensity projected through point (x, y) on the image plane at time t. We assume that the camera aperture is held open during a finite exposure time interval $[0, T]$, during which time object positions and scene lighting conditions may vary.

The goal of a motion blur image synthesis algorithm is to compute a motion blurred image representing the amount of light accumulating on the image plane during the exposure time interval. The result of this computation must be expressed as a discrete set of colour values suitable for display on a digital display device. To this end, we partition the image plane into $X \times Y$ pixels, such that pixel (x, y) covers the set P_{xy} of image plane positions. For each pixel (x, y), we can sample the intensity function through, for example, the pixel centre (c_x, c_y), and form our synthetic image from these sampled intensities.

Since we are considering a finite exposure time and a dynamic scene, the incident intensity $I(c_x, c_y, t)$ seen through the centre of pixel (x, y) may vary over the course of the exposure time interval. What value of t should we use for our intensity sample? Sampling all pixel intensities at a particular time will merely produce an "instantaneous image", as would be produced by a camera with an instantaneous exposure time. To produce a motion blurred image, we need to blend together many such instantaneous images, in order to simulate accumulation of light on the image plane over the entire exposure time interval.

If we consider the intensity function to be a signal, we see that this blending process represents a temporal filtering operation applied to the intensity signal. Instead of sampling the intensity signal I directly, we can convolve it with a temporal filter $h_t(t)$ to produce a motion-blurred signal I' which is then sampled to produce a motion-blurred image.

From the theory of signal processing, we know that temporal filtering removes high temporal frequencies in the intensity signal prior to sampling. If not removed, these high frequencies can result in undesirable temporal aliasing artifacts such as *strobing*. Strobing refers to the effect in which an object appears to suddenly pop into existence at successive positions along its trajectory, as if it were being illuminated by a flashing strobe light. Motion blur serves to smooth out these jerky motions. The exact nature of the smoothing is determined by the choice of temporal filter h_t. The filter may be selected based on its spectral characteristics, or alternatively to simulate the filtering properties of some actual camera aperture, or to achieve a particular artistic effect.

Just as high temporal frequencies give rise to temporal aliasing artifacts in animated sequences, high spatial frequencies result in spatial aliasing artifacts within the individual frames of the animation. These artifacts may manifest themselves in the form of jagged edges, broken highlights, Moiré patterns, or any number of visually objectionable phenomena. For this reason, it is also desirable to filter the intensity signal in space before it is sampled. We combine the temporal filter $h_t(t)$ with a spatial filter $h_s(x, y)$ to form a spatio-temporal filter $h(x, y, t) = h_t(t)\, h_s(x, y)$. Convolving the intensity signal with this filter produces the filtered signal $I'(x, y, t)$, which is then sampled at the pixel centre (c_x, c_y) at the midpoint t_{mid} of the exposure time interval to produce the final filtered pixel intensity I_{xy}:

$$I_{xy} = I'(x_c, y_c, t_{mid}) = \iint\limits_{P_{xy}} \int_0^T h(c_x - x, c_y - y, t_{mid} - t) I(x, y, t)\, dt\, dx\, dy \tag{2.1}$$

More pronounced spatial filtering can be produced by widening the spatial filter function and performing the above spatial integration over an area larger than the actual square pixel area P_{xy}. For the sake of simplicity, we will continue to use P_{xy} to denote the area over which spatial integration is performed.

It is common to factor the intensity function into two terms, a visibility term $G_n(x, y, t)$ and a per-object intensity term $I_n(x, y, t)$, where

$$G_n(x, y, t) = \begin{cases} 1 & \text{if object } n \text{ is visible through image location } (x, y) \text{ at time } t; \\ 0 & \text{otherwise.} \end{cases}$$

and $I_n(x, y, t)$ is the intensity of the point on object n seen through image location (x, y) at time t. Using this decomposition, we see that

$$I(x, y, t) = \sum_{n=1}^{N} G_n(x, y, t) I_n(x, y, t)$$

where N is the number of objects in the scene. Therefore, we can rewrite (2.1) as:

$$I_{xy} = \sum_{n=1}^{N} \iint\limits_{P_{xy}} \int\limits_{T} h(c_x - x, c_y - y, t_{mid} - t) G_n(x, y, t) I_n(x, y, t)\, dt\, dx\, dy \tag{2.2}$$

Computation of the final pixel colours I_{xy} using (2.2) is the goal of all existing motion blur algorithms. Unfortunately, an exact solution is beyond reach in the general case, so some approximation method must be employed. We will examine the approximations used by existing motion blur algorithms.

2.3 Analytic Methods

We classify a motion blur algorithm as *analytic* if it attempts to exactly evaluate the value of the integral (2.2) for each pixel. It is extremely difficult to perform this integration exactly in the general case, since closed-form expressions for the visibility and shading functions are not easily obtained. The algorithms described in this section make simplifying assumptions about the nature of the integrand in order to render the problem solvable by analytic means.

This section describes two analytic motion blur algorithms. Both are similar in that they consider only simple geometry and constrain the motion of this geometry in some way. The main difference between the two algorithms lies in their treatment of spatial antialiasing. The first algorithm, by Korein and Badler [8], ignores spatial antialiasing, while the second algorithm, due to Grant [6], places tighter restrictions on allowed object motions in order to be able to perform spatial antialiasing.

2.3.1 Korein and Badler

This paper constitutes one of the earliest attempts at simulating motion blur in computer graphics. The algorithm considers only a single sample position for each pixel (namely the pixel centre), and computes a temporally filtered intensity value for that position. The algorithm does not perform spatial antialiasing.

Formulation

The algorithm computes pixel intensities using the following formulation, which is simply (2.2) without the spatial integration:

$$I_{xy} = \sum_{n=1}^{N} \int_{0}^{T} h_t(t_{mid} - t) G_n(x_c, y_c, t) I_n(x_c, y_c, t)\, dt \tag{2.3}$$

Since only one object is visible through the pixel centre at any given time, the authors note that it is possible to partition the exposure time interval $[0, T]$ into "visible intervals" T_n during which each object n is visible. More formally, $t \,\epsilon\, T_n$ if and only if $G_n(x_c, y_c, t) = 1$. The integral (2.3) can now be decomposed as follows:

$$I_{xy} = \sum_{n=1}^{N} \int_{T_n} h_t(t_{mid} - t) I_n(x_c, y_c, t)\, dt \tag{2.4}$$

If we ignore shading variations and consider only constant-coloured objects, then the per-object intensity function is a constant, say $I_n(x, y, t) = C_n$. The integral then becomes:

$$I_{xy} = \sum_{n=1}^{N} C_n \left(\int_{T_n} h_t(t_{mid} - t)\, dt \right) \tag{2.5}$$

Since an analytic description of the filter function h_t is usually available, all that needs to be done to compute (2.5) is to compute the visible intervals T_n, and to integrate the filter function over these intervals. The authors describe how to compute visible intervals for spheres and for polygonal geometry.

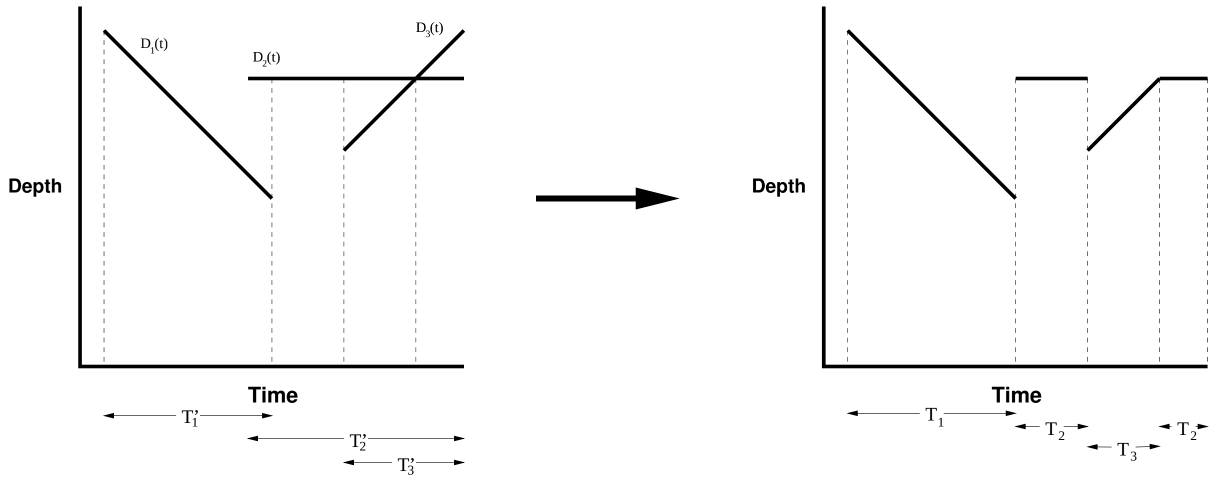


Figure 2.1: Computing the visible intervals in Korein and Badler's algorithm. First, the covering intervals T'_n are computed and the linear depth functions synthesized, as in the figure at left. A two–dimensional hidden surface removal is then performed on this graph to determine the visible intervals T_n during which each object is visible, as shown in the graph at right.

Computing Visible Intervals

To compute the visible interval for object n, the authors first compute a "covering interval" T'_n during which the screenspace projection of the object covers the pixel centre, ignoring possible occlusion by other objects. A linear depth function $D_n(t)$ is then formulated that specifies the distance to the object (as seen through the pixel centre) as a function of time. The depth functions are then used resolve visibility and determine the visible intervals for each object. Figure 2.1 illustrates the process.

For spheres, the covering interval is determined by intersecting the pixel centre with the silhouette of the sphere in screen space. For polygonal geometry, covering intervals are computed in 3d camera space by intersecting the ray through the pixel centre with each edge of the polygon and combining the results. For both types of geometry, the depth function is constructed by linearly interpolating the distances to the object computed at the endpoints of the covering intervals.

Assessment

This paper provides a useful first step towards solving the motion blur problem, but it has its shortcomings. The authors note that the algorithm's inability to perform spatial antialiasing or to handle complex shading variations across visible intervals are handicaps that limit its utility in general situations. Furthermore, in order to compute covering intervals for polygonal geometry, the algorithm requires that all vertices of a polygon have identical velocities in camera space. Thus, the algorithm cannot handle anything but simple

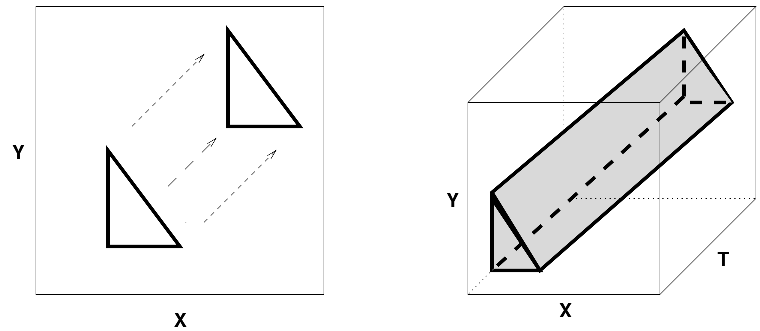


Figure 2.2: Screenspace object motion and resulting covering volume.

translational motion of polygons.

2.3.2 Grant

Grant's approach attempts to extend the ideas first proposed by Korein and Badler [8] to include spatial antialiasing. In considering two extra spatial dimensions, Korein and Badler's temporal "visible intervals" become 3D spatio-temporal "visible volumes" over which the filter function must be integrated.

Formulation

Grant's algorithm is simply the three-dimensional analogue of Korein and Badler's one-dimensional approach. When spatial dimensions are considered, the temporal covering intervals T'_n become spatio-temporal covering volumes V'_n, where a spacetime point $(x, y, t) \epsilon V'_n$ if object n projects to image location (x, y) at time t. Figure 2.2 shows the covering volume swept out by the linear motion of a polygon in screenspace.

It may be the case that not all points in the covering volume V'_n correspond to unoccluded projections of polygon n. We denote by $V_n \subseteq V'_n$ the "visible volume" corresponding to unoccluded projections of object n, i.e. $G_n(x, y, t) = 1$ for all $(x, y, t) \epsilon V_n$. Since only one polygon is visible at a particular image location at a particular time, these visible volumes partition the integration domain of (2.2), hence the motion blur integral (2.2) can be rewritten as:

$$I_{xy} = \sum_{n=1}^{N} \iiint\limits_{V_n} h(x_c - x, y_c - y, t_{mid} - t) I_n(x, y, t)\, dt\, dx\, dy \tag{2.6}$$

Like Korein and Badler [8], Grant only considers objects of constant intensity, with object n having intensity

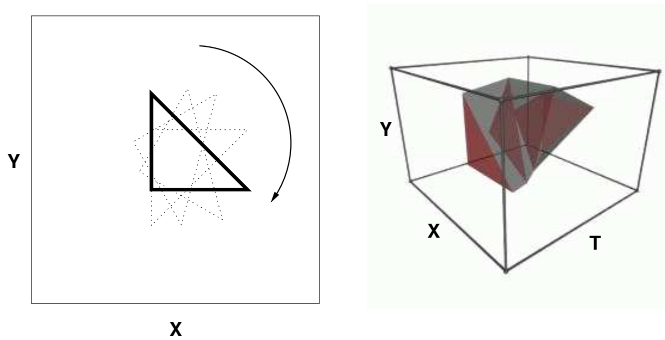


Figure 2.3: Rotating polygon and 3-polyhedron approximation of its covering volume.

C_n. This permits further simplification of (2.6) to:

$$I_{xy} = \sum_{n=1}^{N} C_n \left(\iiint_{V_n} h(x_c - x, y_c - y, t_{mid} - t)\, dt\, dx\, dy \right) \tag{2.7}$$

For each object, Grant's algorithm first computes the visible volume V_n, then passes this volume to a filtering algorithm that integrates the filter function over the volume.

Computing and Filtering Visible Volumes

Grant's approach considers only polygonal geometry, and requires that all moving polygon vertices follow a linear motion path in screenspace. Any nonlinear vertex motion must be replaced by a piecewise linear approximation. In this manner, the covering volume swept out by a moving polygon is a collection of one or more spacetime polyhedra. Figure 2.3 shows a rotating polygon and a 3-polyhedron approximation of the resulting covering volume.

To isolate the visible volume $V_n \subseteq V_n'$, Grant tests V_n' for intersection with other covering volumes. If V_n intersects no other covering volumes, then object n is always visible, hence $V_n = V_n'$. Where covering volumes intersect, a depth function $D_n(x, y, t)$ is used to resolve visibility in the intersecting volume. Grant assumes that the intersection of two visible volumes (V_n' and V_m', say) can be split by a plane (which is an approximation, since this is not generally the case) into one part belonging to V_n and one part belonging to V_m. In this way, the visible volumes produced by the algorithm are always polyhedral.

Once the visible volumes have been computed, all that remains to evaluate I_{xy} using (2.7) is to integrate the filter function $h(x, y, t)$ over each visible volume. Grant uses a three-dimensional extension of the analytic filtering algorithm of Feibush, Levoy, and Cook [5]. Assuming a radially symmetric filter function, the algorithm works by precomputing the filter function integral over various right tetrahedra centred at the spacetime pixel centre. By decomposing the visible volume into sums and differences of such tetrahedra, a nearly exact filtered result can be calculated by combining the corresponding precomputed values.

Assessment

Grant's algorithm is significant in that it is the only analytic algorithm that addresses both spatial and temporal filtering. Unfortunately, it constrains the problem too much to be of general use. The requirement of piecewise linear motion means that complex object motion can only be approximated by large collections of polyhedral covering volumes, all of which must be processed by the visibility resolution and filtering algorithms. Furthermore, the assumed radial symmetry of the filter function precludes the use of arbitrary combinations of spatial and temporal filters (a temporal box filter could not be combined with a Gaussian spatial filter, for example). Finally, the constant shading assumption renders the algorithm useless in the presence of interesting shading algorithms.

2.4 Monte Carlo Integration Methods

The analytic algorithms described in the previous section produce good results, but only under the assumption of constant-coloured polygonal geometry moving linearly. This assumption is too restrictive for general production environments, where complex procedural shaders, non-polygonal geometry, and nonlinear motion frequently combine to produce an intensity function that is too complex to be analytically integrated.

Regardless of its complexity, however, we can always sample the intensity function at any particular point in space and time. Given this, we can employ the technique of *Monte Carlo integration* to approximate the value of the integral (2.2). In this context, Monte Carlo integration techniques describe methods for constructing a statistical estimate of the value I_{xy} using weighted combinations of randomly distributed point samples of the integrand.

We first discuss some basic concepts of Monte Carlo integration, including the variance reduction techniques of stratified sampling and importance sampling. We then discuss two motion blur algorithms based on Monte Carlo integration, Distribution Ray Tracing [4] and the Accumulation Buffer [7].

2.4.1 Monte Carlo Integration

Say we wish to evaluate an integral of the form

$$I = \int_a^b f(x)\,dx \tag{2.8}$$

If we consider a set $\{X_1, \ldots, X_N\}$ of independent, identically distributed random variables uniformly distributed on the interval $[a, b]$, then the sum

$$\hat{I} = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i) \tag{2.9}$$

is an *unbiased estimator* of the value I, meaning that on average, a sample of $\hat{I}$ will have the value I (mathematically, we say that the expectation $E[\hat{I}] = I$).

It is important to note that $\hat{I}$, being a sum of random variables, is itself a random variable. Even though it has the desired value I on average, it is possible for individual samples of $\hat{I}$ to deviate widely from this mean. This variability is quantified by the *variance* of $\hat{I}$, denoted by $V[\hat{I}]$. Several *variance reduction techniques* may be employed to improve the quality of the estimator. We discuss three of the most common techniques: stratified sampling, importance sampling, and importance-stratified sampling.

Stratified Sampling

The X_i described above generate sets of random sampling patterns. If successive patterns tend to concentrate samples in different parts of the domain, then the resulting estimator will have a high variance. By partitioning the integration domain into strata and casting a fixed number of samples into each, stratified sampling ensures that each generated pattern covers the domain uniformly.

Consider a partition of the integration domain $[a, b]$ into strata $\{S_1, ..., S_k\}$, with stratum S_i having width v_i. By distributing n_i samples uniformly randomly over each S_i, we can construct a Monte Carlo estimator $\hat{I}_i$ for the true integrated value $I_i = \int_{S_i} f(x)dx$:

$$\hat{I}_i = \frac{v_i}{n_i} \sum_{j=1}^{n_i} f(X_{i,j}) \tag{2.10}$$

where $X_{i,j}$ is a random variable corresponding to the j^{th} sample in S_i. These individual estimators can be combined to form an estimator $\hat{I}_{stratified}$ for the value of the integral over the whole domain:

$$\hat{I}_{stratified} = \sum_{i=1}^{k} \hat{I}_i \tag{2.11}$$

Veach [13] shows that the variance of this stratified estimator is minimized when the strata are selected such that the true integrated values I_i vary as much as possible between strata. When no *a priori* information about the integrand is available, however, it is best to use equal-volume strata, each of which receives a single sample. This strategy maximizes the probability of obtaining strata having widely varying I_i's.

Importance Sampling

Our original estimator $\hat{I}$ used sample positions uniformly distributed over the domain of integration. In fact, we can construct an unbiased estimator based on any sampling distribution. If the X_i are independent identically distributed random variables with probability density function (PDF) $p(x)$, then

$$\hat{I}_{importance} = \frac{b-a}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{2.12}$$

is an unbiased estimator of I. If we have prior knowledge of the shape of the integrand, then we can reduce the variance of the estimator by choosing a $p(x)$ that approximates this shape. This technique, known as *importance sampling*, tends to concentrate samples in "important" regions of the domain, i.e. regions where the integrand is large.

Samples can be drawn from an arbitrary distribution $p(x)$ using the *inversion method.* Given a uniformly distributed random variable T on $[0, 1]$ and the cumulative distribution function (CDF) $P(x) = \int_a^x p(x)\, dx$, the random variable $P^{-1}(T)$ has PDF $p(x)$.

Importance-Stratified Sampling

Although importance sampling on average tends to place samples where they are most needed, it is still possible for importance sampling patterns to occasionally clump points together in an unimportant region of the domain of integration. To prevent this, importance sampling can be combined with stratified sampling.

Given an importance distribution $p(x)$, *importance-stratified sampling* chooses strata of equal volume v_i under $p(x)$, and distributes samples in each stratum S_i according to the PDF $p_i(x)$, where

$$p_i(x) = \begin{cases} \frac{p(x)}{v_i} & \text{if } x \,\epsilon\, S_i \\ 0 & \text{otherwise.} \end{cases} \tag{2.13}$$

The importance-stratified estimator is constructed as:

$$\hat{I}_i = \frac{v_i}{n_i} \sum_{j=1}^{n_i} \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{2.14}$$

$$\hat{I}_{importance-stratified} = \sum_{i=1}^{k} \hat{I}_i \tag{2.15}$$

Importance-stratified sampling patterns can be generated by a simple modification of the inversion

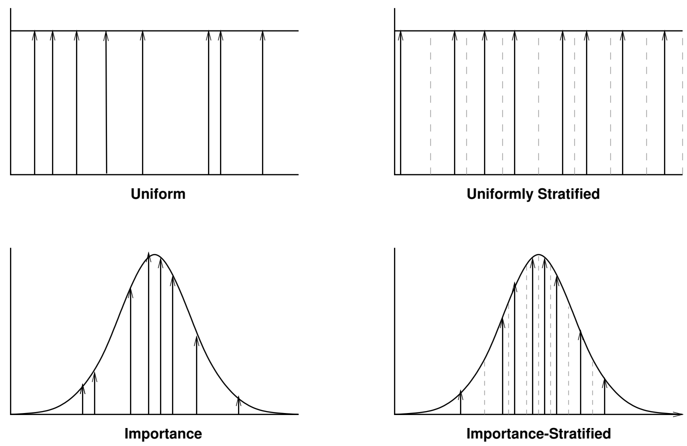


Figure 2.4: Uniform, Stratified, Importance, and Importance-stratified sampling patterns. In stratified versions, a fixed number of samples (1 in this case) are taken in each stratum. The dashed lines denote boundaries of strata, and the bold arrows denote sample positions.

method. If we stratify the uniform random variable T into equal-volume strata, then $P^{-1}(T)$ will generate an importance-stratified pattern. Figure 2.4 shows several sampling patterns constructed using the techniques described in this section.

2.4.2 Distribution Ray Tracing

Distribution Ray Tracing [4] was the first algorithm to use Monte Carlo integration to solve the motion blur integral (2.2). The one-dimensional integration techniques described in the previous section extend naturally to handle this three-dimensional spacetime integral. In fact, the authors extend the approach to even more dimensions to simulate other "fuzzy" phenomena such as soft shadows, depth–of–field, surface gloss, and translucency. For the sake of clarity, we will ignore these added dimensions and concentrate solely on spatio-temporal filtering.

Formulation

The Distribution Ray Tracing algorithm uses a variant of importance-stratified Monte Carlo integration to approximate the integral (2.2) for each pixel. First, the the pixel area is importance-stratified into spatial strata $\{S_1, ..., S_N\}$ using the spatial filter function $h_s(x, y)$ as an importance function. Similarly, the temporal filter function $h_t(t)$ is used to importance–stratify the exposure time interval into temporal strata $\{T_1, ..., T_N\}$. The Cartesian product of the spatial and temporal strata form an importance–stratification $\{S_1, ..., S_N\} \times \{T_1, ..., T_N\}$ of the spacetime pixel.

Note that the above scheme produces N^2 spacetime strata. As N increases, it becomes prohibitively

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Correlated

7	11	3	14
4	15	13	9
16	1	8	12
6	10	5	2

Uncorrelated

Figure 2.5: Spatio-temporal sampling patterns for Distribution Ray Tracing. Each pattern shows the temporal stratum $\pi(i)$ associated with each of 16 spatial strata resulting from a 4×4 subdivision of the pixel area. The pattern on the left associates earlier time strata ($\pi(i) \epsilon \{1, \ldots, 8\}$) with the spatial strata on the left of the pixel, and later time strata ($\pi(i) \epsilon \{9, \ldots, 16\}$) with the spatial strata on the right. In the second pattern, no such correlation between spatial and temporal strata is apparent.

expensive to place a sample in each stratum. Instead, only N strata $(S_i, T_{\pi(i)})$ are selected, where $\pi(i)$ represents a permutation of the integers from 1 to N. A single sample is cast into each of these strata. The resulting spatio-temporal sample distribution is reasonably well distributed in that its temporal projection places a single sample in each temporal stratum, and its spatial projection places a single sample in each spatial stratum.

The permutation $\pi(i)$ is selected to avoid correlation between spatial and temporal sample locations, as this can bias the estimator. For example, the first pattern shown in Figure 2.5 only samples the left side of the pixel during the first half of the exposure time interval and the right half of the pixel during the last half of the interval. A polygon moving from right to left across the pixel could be completely missed by such a pattern. By removing this correlation, the second pattern stands a better chance of accurately sampling arbitrary object motion through the pixel.

Once the N sample positions $X_i = (x_i, y_i, t_i)$ have been computed, the final pixel intensity is computed as:

$$I_{xy} = \frac{1}{N} \sum_{i=1}^{N} \frac{I(x_i, y_i, t_i) h(x_c - x_i, y_c - y_i, t_{mid} - t_i)}{h(x_c - x_i, y_c - y_i, t_{mid} - t_i)} = \frac{1}{N} \sum_{i=1}^{N} I(x_i, y_i, t_i) \tag{2.16}$$

Equation (2.16) is derived from the importance-based estimator in equation (2.12) rather than from the importance-stratified estimator in (2.15). The reason for this is that Distribution Ray Tracing does not employ a true importance-stratified estimator (since not all strata are sampled). By ignoring all but N strata, Distribution Ray Tracing is simply using the concept of importance-based stratification to construct a particularly well-distributed importance-based estimator.

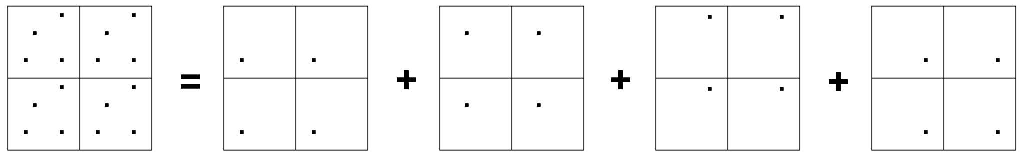


Figure 2.6: Replicated random pattern as a sum of uniform grids.

Assessment

Distribution Ray Tracing has become quite popular due to its simplicity and the fact that it can simulate many "fuzzy" phenomena in addition to performing spatio-temporal antialiasing. At low sampling densities, however, the images produced by the algorithm tend to be quite noisy. Furthermore, the predefined stratification of the temporal domain can cause banding artifacts in motion-blurred images due to the presence of discontinuities in the integrand.

2.4.3 The Accumulation Buffer

Throughout the history of computer graphics, there have always been two largely distinct streams of rendering research; offline algorithms that simulate sophisticated effects such as motion blur (albeit at high computational cost), and hardware-based algorithms that can deliver comparatively low-quality images, but at interactive rates. The Accumulation Buffer represents one of the first bridges between these two streams, allowing high-quality offline rendering algorithms to take advantage of powerful graphics hardware.

The Accumulation Buffer itself is simply a high-precision auxiliary graphics buffer into which multiple rendered images can be accumulated. High-quality renderings are achieved by accumulating and blending together many separate low-quality images.

Formulation

In general, modern graphics hardware only samples the intensity function on a uniform grid of pixel centres, thus the nonuniform random pattern required by (2.16) cannot be sampled in a single hardware rendering pass. Instead, the nonuniform pattern must be decomposed into uniform grids that can be sampled by the hardware.

Assuming that the random pattern $\{X_1, ..., X_N\}$ within a (spatio-temporal) pixel is replicated over the entire image (i.e. used to sample every pixel in the image), then the overall sampling pattern can be expressed as the sum of N uniform sampling grids, each slightly offset from the other. Figure 2.6 demonstrates this on a two-dimensional spatial pattern replicated over 4 pixels.

Given this realization, we can think of an image $I^{(i)}$ produced by sampling the intensity function on the uniform grid corresponding to the i^{th} sample position. The summation (2.16) over sample positions can then be rewritten as a summation over these images as:

$$I_{xy} = \frac{1}{N} \sum_{i=1}^{N} I_{xy}^{(i)} \tag{2.17}$$

To generate the image $I^{(i)}$ corresponding to sample position $X_i = (x_i, y_i, t_i)$ we simply move all objects to their positions at time t_i, modify the viewing transformation to align the pixel centres (the canonical sample positions for hardware-based rendering) with the appropriate grid, and render the image using hardware acceleration. Each image is rendered and added to the contents of the accumulation buffer. Once all images have been rendered, the intensities stored in the accumulation buffer are scaled by $\frac{1}{N}$, and the finished result is transferred back to the display buffer.

Assessment

Compared with software-based Distribution Ray Tracing, the hardware-accelerated accumulation buffer algorithm sacrifices flexibility and image quality for speed. While software implementations of Monte Carlo sampling typically generate new random sample patterns for each pixel, the hardware-based algorithm is forced to use a single pattern, replicated over all pixels. The resulting pattern is locally stochastic but globally periodic (it is, after all, just the sum of several uniform sampling grids), which can sometimes cause large-scale aliasing artifacts in images. Another drawback to this approach is that it can only blur those shading effects that can be simulated by graphics hardware. Procedural textures in particular are easily added to software-based renderers, but can be quite tricky to implement using graphics hardware.

2.5 Postprocess Motion Blur Methods

The analytic and Monte-Carlo-based algorithms discussed in the previous two sections produce motion blur by analytic filtering or point sampling of the full three-dimensional spacetime intensity function. These approaches are capable of producing correctly filtered images, but at considerable computational cost, especially in the context of geometrically complex scenes.

When correctness is not essential, *postprocess motion blur algorithms* can often be used to generate aesthetically pleasing images at a fraction of the cost of other algorithms. Postprocess algorithms operate by "smearing" an instantaneous image of an object along that object's trajectory in the image. Because the smearing is done as a postprocess in two-dimensional image space, these algorithms are fast and independent of the rendering algorithm used to generate the instantaneous image.

We discuss four different postprocess motion blur algorithms. Potmesil and Chakravarty [12] were the first to suggest such an algorithm, devising an approach that can be applied in the Fourier domain. Max [10] described an efficient image-space implementation of this approach, then extended it to handle more general motion of polygonal meshes [9]. Finally, Chen [2] applied his view-interpolation algorithm to generate motion blur caused by camera motion.

2.5.1 Potmesil and Chakravarty

This paper describes an algorithm for adding motion blur to an instantaneous image by convolution with a filter derived from the motion of an object. The authors first describe the algorithm for blurring a single object; then they extend the algorithm to handle multiple objects.

Formulation

The authors formulate the approach in the context of a single object moving against a black background. The essential realization at the heart of all postprocess motion blur algorithms is that for a particular object n moving across a pixel (x, y), the standard temporal motion blur integral

$$I_{xy} = \int_0^T h_t(t) I_n(x_c, y_c, t)\, dt \tag{2.18}$$

can be recast as an integral over the surface of the object:

$$I_{xy} = \int_0^T h(t) I_n(r(t))\, dt \tag{2.19}$$

where $r(t)$ is the line on the surface traced out by the ray through the pixel centre over the exposure time interval, and $I_n(r(t))$ denotes the object's intensity along this line. Given an instantaneous image of the object at time t^*, this integral can be approximated in screenspace as

$$I_{xy} \approx I_{xyt^*} = \int_0^T h(t) I(r^*(t))\, dt \tag{2.20}$$

where $r^*(t)$ is the projection of $r(t)$ into the image plane at time t^*, and $I(r^*(t))$ is the image intensity along $r^*(t)$ at time t. Given a description of the screenspace motion of the object, the path $r^*(t)$ can be computed for each pixel centre. Figure 2.7 illustrates the relevant object-space and image-space quantities.

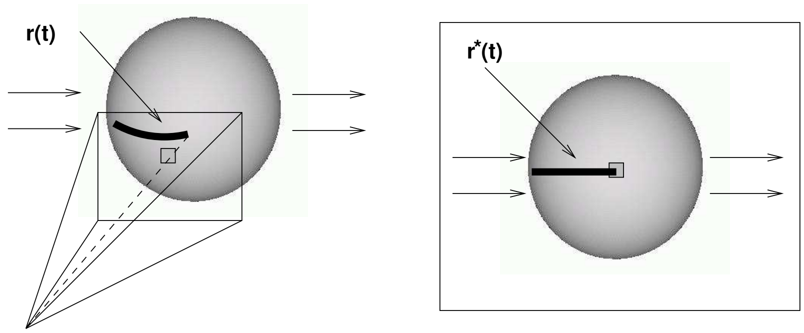


Figure 2.7: Path traced out by a pixel centre in object space and in image space.

The authors show that the integral 2.20 is simply a convolution of the instantaneous image with a *point spread function* (PSF) derived from the projected motion path $r^*(t)$. If all points in the image follow the same path (i.e. $r^*(t)$ is the same for all points in the image), then the PSF is *space-invariant*, and this convolution can be carried out in the frequency domain. First, the Fourier transforms of the image and the motion PSF are computed and multiplied together, then the inverse Fourier transform is applied to produce the final motion-blurred image.

Multiple Objects

When multiple objects moving in different directions are present in the image, then the resulting PSF is no longer space-invariant, thus the frequency domain algorithm cannot be applied. To deal with this more complicated case, the authors suggest that each individual object be blurred separately using the basic algorithm, and the resulting per-object blurred images be composited in back-to-front depth order to form the final image.

To accomplish this compositing, an additional instantaneous mask image is generated for each object to mark the pixels covered by the object. The blur algorithm is applied to this mask to produce a blurred mask that is then used to composite the blurred object image with a background image. The blurred mask I^{mask} approximates the object's temporal coverage of each pixel:

$$I_{xy}^{mask} \approx \int_0^T G(x_c, y_c, t)\, dt \tag{2.21}$$

With the mask computed, the compositing operation can be expressed as

$$I_{xy} = I_{xy}^{mask} I_{xy}^{front} + (1 - I_{xy}^{mask}) I_{xy}^{background} \tag{2.22}$$

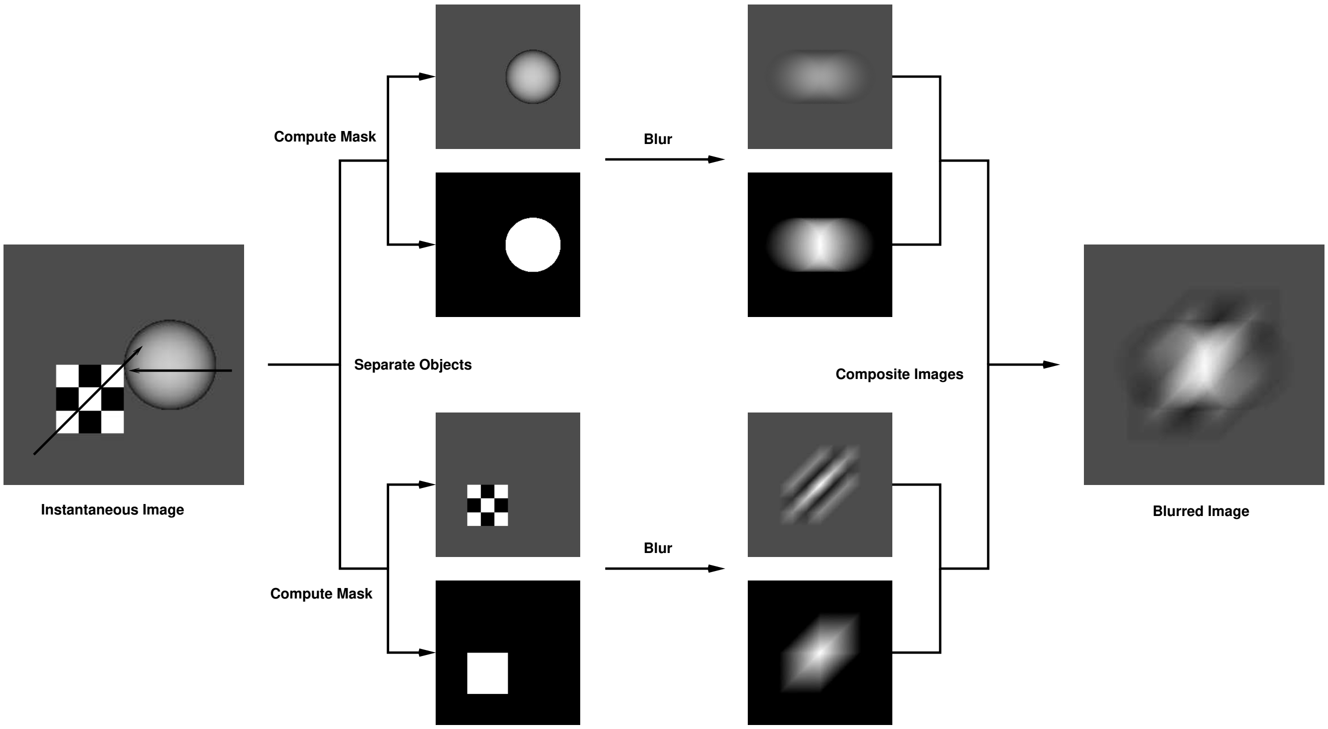


Figure 2.8: Blurring and compositing of individual object images and masks in Potmesil and Chakravarty's algorithm.

where I_{xy}^{front} denotes the intensity of the blurred foreground object image, and $I_{xy}^{background}$ denotes the intensity of the background image. By successively compositing objects into the background image in back-to-front order, the final image is obtained. Figure 2.8 illustrates this process.

Assessment

When its preconditions are satisfied, this algorithm is capable of quickly generating pleasing images. Unfortunately, these preconditions are violated quite often. The main problem is that the paper's frequency-space evaluation of equation (2.20) is only correct when $r^*(t)$ is identical for every point in the image (a condition known as *uniform motion*). Treating each object individually eliminates nonuniform motion resulting from multiple objects moving in different directions, but the algorithm still assumes that all points on a particular object are moving uniformly. In practice, perspective distortion and complex object movements (rotation and scaling, for example) often cause this assumption to be violated. Although the authors state that a wide variety of nonuniform motions can be made uniform by a transformation, they give no general algorithm for computing such a transformation.

Another problem with this algorithm is that in order to handle multiple objects, it requires that the objects be sorted into non-overlapping depth layers. Intersecting objects or objects whose depth relationship changes over the course of the exposure time interval will often not be properly blurred by this algorithm.

2.5.2 Max (1985)

One drawback of Potmesal and Chakravarty's [12] approach is that it requires frequency-space computations to create the motion-blurred image. In this paper, Max describes an efficient *raster blur algorithm* for computing equation (2.20) in image space for simple linear object motions.

Formulation

If all points on an object are moving horizontally with screenspace velocity $(dx, 0)$, then the integral (2.20) can be approximated by summing along a row of pixels in the instantaneous image I^{inst}:

$$I_{xy} = \sum_{k=0}^{Tdx} I^{inst}_{x-k,y} \tag{2.23}$$

Since all image points are weighted equally, this represents a temporal box filter. This summation can be computed incrementally, since

$$I_{x+1,y} = I_{xy} + I^{inst}_{x+1,y} - I^{inst}_{x-Tdx-1,y} \tag{2.24}$$

This fast algorithm can be adapted to handle non-horizontal uniform linear motion by a three-step "skew-blur-unskew" process. First, a nonlinear skew transformation is applied to the instantaneous image that aligns the screenspace velocity vector with the horizontal axis. Then, the resulting skewed image is blurred horizontally using the fast incremental algorithm. Finally, the skewed blurred image is unskewed. For an image-space velocity vector (dx, dy), the skew transformation can be expressed as:

$$(x, y) \to (x, y - x\frac{dy}{dx}) \tag{2.25}$$

Some local averaging may be needed to compensate for the fact that pixel centres are not always projected onto other pixel centres by this transformation. Figure 2.9 illustrates this "skew-blur-unskew" process. Max handles multiple objects in the same way as Potmesil and Chakravarty, by computing blurred coverage masks and using them to composite blurred object images in back-to-front depth order.

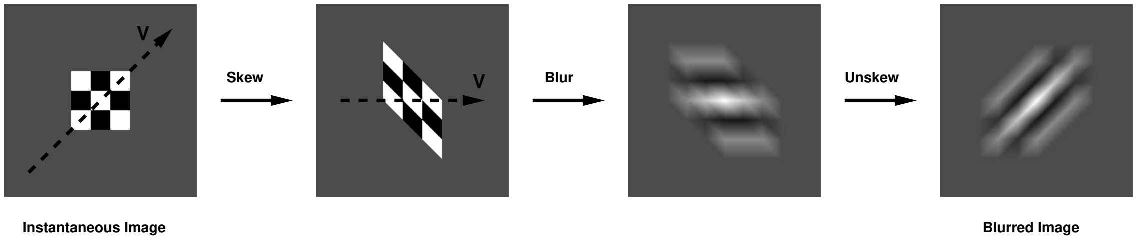


Figure 2.9: Max's "skew-blur-unskew" image-space blur algorithm.

Assessment

As Max's algorithm is essentially identical to that of Potmesil and Chakravarty [12], it suffers from the deficiencies inherent in that algorithm. Furthermore, the local averaging involved in the skewing and unskewing may cause overblurring. Nonetheless, this algorithm has been widely implemented in commercial systems due to its simplicity and speed.

2.5.3 Max (1990)

In this paper, Max attempts to apply his raster blur algorithm to nonuniform motion of polygonal meshes. Under non-uniform motions such as rotations, each polygon in the mesh will have a different image space motion vector, so the mesh cannot be processed as a single object by the basic raster blur algorithm. Max proposes two solutions to handle this case, the *transformation method* and the *sum of weights method*.

Transformation Method

If the instantaneous image of the mesh can be warped to align the motion vectors of the individual polygons, then the object motion is uniform in the distorted image space. The warped image can be blurred using the raster blur algorithm, and the result unwarped to produce an image whose motion blur is consistent with the original non-uniform motion.

Finding an invertible transformation to accomplish the desired warping is not trivial, however. For rotational motions, Max notes that a conversion to polar coordinates is sometimes sufficient, but he gives no algorithm for computing the transformation in general.

Sum of Weights Method

When the motion of the mesh cannot be transformed into uniform motion, each polygon in the mesh must be blurred separately. For each visible mesh polygon, a single motion vector is computed as the average of the motion vectors of the polygon vertices. Treating each polygon as a separate object, a blurred image can

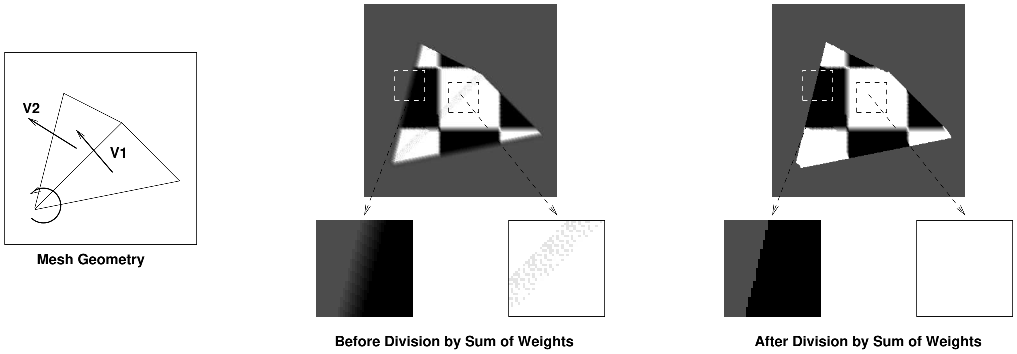


Figure 2.10: Max's "sum of weights" image-space blur algorithm.

then be computed for each as in the Max's earlier multi-object algorithm [10]. After computing the blurred object and mask images I^i and M^i respectively, the final image is computed by:

$$I_{xy} = \sum_{i=1}^{N} I_{xy}^i \tag{2.26}$$

Because each mesh polygon moves independently, it is possible for adjacent polygons to overlap or spread apart as they move, resulting in undesirable artifacts in the blurred image. These problems occur along mesh edges between faces, where they can be detected by examining the "sum of weights" $W_{xy} = \sum_{i=1}^{N} M_{xy}^i$. Where adjacent polygons overlap, the sum of weights $W_{xy} > 1$, and darker-than-expected pixels result. Similarly, lighter-than-expected pixels appear when $W_{xy} < 1$, namely along edges between polygons that are moving apart.

Max's solution to this problem is to simply divide the pixel intensities computed by (2.26) by the sum of weights W_{xy}, thereby evening out these overcontributions and undercontributions. The resulting pixel colours can be expressed as:

$$I_{xy} = \frac{\sum_{i=1}^{N} I_{xy}^i}{\sum_{i=1}^{N} M_{xy}^i} \tag{2.27}$$

Max notes that this normalization works well for points inside the mesh, but produces artifacts along the moving silhouette of the mesh, where it essentially negates the motion blur effect.

Figure 2.10 demonstrates this algorithm applied to a rotating mesh consisting of two triangles. The middle image was produced by blurring the individual triangles along directions V_1 and V_2, and combining the images using equation (2.26). The zoomed areas demonstrate the motion blurred silhouette edge and the undercontribution artifacts along the edge joining the triangles. The rightmost image demonstrates the result of dividing by the sum of weights, as in Equation 2.27. The zoomed areas show that the undercontribution artifacts are eliminated, but the silhouette motion blur is ruined.

Assessment

Because of the silhouette problem described above, this algorithm is only useful for blurring polygonal meshes whose silhouette edges are either not visible or are not moving in the image. From an efficiency standpoint, the algorithm is somewhat undesirable because in processing mesh polygons individually, it introduces a dependency on the geometric complexity of the scene. Still, it is cheaper than analytic or sampling methods, and may be the only option available when speed is of the essence and the object motion is too complex for the basic raster blur algorithm.

2.5.4 Chen

Chen developed an image-based algorithm for creating novel views of a scene by interpolating several reference images of the scene taken from different viewpoints. Camera fly–by animations can be created by constructing interpolated images for various points along the motion path of the camera. In general, this is much quicker than re-rendering the scene from each successive viewpoint.

Motion blur can be incorporated into these camera animations by simply blending together many images taken from slightly different viewpoints along the camera path. This is a particularly difficult case for most motion blur algorithms, as camera motion usually causes most (if not all) objects in the scene to move simultaneously in the image plane.

Formulation

Given two images A and B of a scene taken from viewpoints V_A and V_B, Chen's algorithm constructs approximate images of the scene taken from viewpoints along the line joining V_A and V_B. Images A and B store colour, depth, and a 3d camera-space motion vector at each pixel. Using this information, the algorithm constructs a 2d *morph-map* that determines the image-space movement of the 3d points visible through each pixel in image A. By moving each pixel along its "offset vector" stored in the map, image A is morphed into an approximation of image B. Similarly, an inverse morph map is computed from image B to image A. To generate an image representing a viewpoint midway between images A and B, the algorithm morphs image A halfway to image B, morphs image B halfway towards image A, then blends the two images together.

To generate a motion blurred image resulting from camera movement from V_A to V_B, Chen simply uses the morphing algorithm to generate a series of interpolated images along the camera motion path over the exposure time interval. These images are then blended together to create a motion-blurred image. This approach amounts to supersampling the temporal dimension.

Assessment

Chen's method is one the better image-based motion blur algorithms, as it is the only such algorithm that actually incorporates information taken from different times in the exposure time interval. Occlusion changes and changes in object appearance over the exposure interval are all considered in constructing the blurred image. All other image-based motion blur algorithms must try to extrapolate this information from a single instantaneous image.

This increased quality comes at a cost, however. The algorithm requires at least 2 instantaneous images to generate a motion-blurred picture. The algorithm must also resolve occlusion changes between pixels, and deal with potential holes in the image due to the fact that the morph-maps between two images are not one–to–one. Furthermore, since the algorithm amounts to supersampling in time, it will suffer from temporal aliasing artifacts if too few images are blended.

2.6 Geometric Substitution Methods

Serveral motion blur algorithms are based on the idea of substituting a specially constructed stationary object in place of a moving object. The stationary object is constructed so that an instantaneous rendered image of the stationary geometry approximates a motion blurred image of the moving geometry. Once the "stationary substitute" has been constructed, any static rendering or visiblity determination algorithm can be used to render the motion-blurred image.

Two such algorithms appear in the literature. The first, due to Catmull [1], spatially filters a stationary substitute created by compressing the moving object along its trajectory. The second, due to Wloka and Zeleznik [14], instead uses an extrusion of the moving object along its trajectory, and applies a transparency map to express temporal coverage.

2.6.1 Catmull

Catmull describes an algorithm designed to quickly render large numbers of very small polygons. Within a pixel, visibility determination is performed using a depth sort, and spatial antialiasing of the resulting polygon fragments is achieved using the analytic algorithm of Feibush, Levoy, and Cook [5]. Catmull extends this algorithm to handle motion blur by shrinking polygons along their image-space trajectories prior to visibility

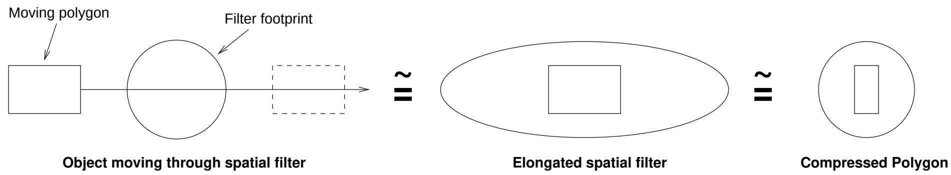


Figure 2.11: Catmull's algorithm (reproduced from Catmull [1]).

determination and filtering.

Formulation

Catmull's algorithm uses purely spatial filtering to approximate spatio-temporal filtering. If the spatial filter for a particular pixel is elongated along the image–space trajectory of a moving polygon, then the resulting spatial integration approximates the true spatio–temporally filtered result for that pixel. To avoid having to elongate the filter, an equivalent approach is to compress the polygon along its trajectory prior to filtering. Catmull uses the latter approach, since Feibush's filtering algorithm [5] requires a circularly symmetric filter kernel. Figure 2.11 illustrates the general idea behind this approach.

The algorithm processes each pixel independently. Once all polygons intersecting a pixel have been compressed along their trajectories, the visibility determination algorithm is invoked on the transformed polygons, and the resulting fragments are filtered analytically.

Assessment

This algorithm has a few drawbacks. First, it can only handle constant-coloured polygons due to its dependence on Feibush's algorithm for spatial filtering. This is not a problem in the context of the Reyes architecture [3], where micropolygons are assigned constant colours prior to visibility determination and filtering, but it does limit the applicability of the algorithm to other rendering architectures. Another drawback is that the visibility determination algorithm only approximates the correct solution for moving objects. It does not properly handle cases where moving objects occlude stationary objects and vice versa.

This algorithm has not found widespread popularity, possibly due to these flaws, but more likely due to the fact that it was introduced concurrently with Distribution Ray Tracing, which solves the problem of spatio-temporal filtering in a much more elegant and extensible manner.

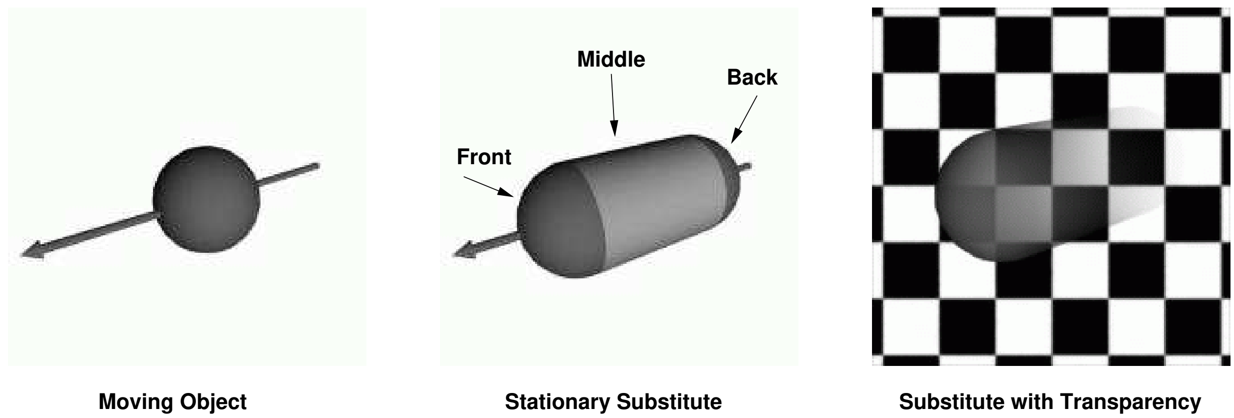


Figure 2.12: Wloka and Zeleznik's algorithm.

2.6.2 Wloka and Zeleznik

Wloka and Zeleznik describe an algorithm in which the object-space volume swept out by a moving object during the exposure time interval (the object's *motion volume*) is used as a stationary substitute for the moving object. The motion volume is rendered with a transparency map that simulates temporal filtering of the moving object. Standard graphics hardware can be used to render the substitute, yielding an algorithm that runs at near-interactive rates.

Formulation

The algorithm approximates the motion volume of an object by first splitting it in half, with the "front" and "back" halves consisting of the parts of the object facing into and away from the direction of motion, respectively. For polyhedral objects, this is accomplished by splitting the object along the edges that form its silhouette as seen through the direction of motion.

Once the object has been split, the motion volume is constructed by moving the back half to its position at the shutter open time, moving the front half to its position at the shutter close time, and constructing quadrilaterals that join corresponding edges on the two halves.

A transparency map derived from a particular temporal filter is then constructed and applied to the motion volume. The transparency of a point on the volume is determined by its relative position along the axis of motion. The authors suggest that a linear filter approximation that assigns decreasing transparencies to points toward the back of the volume produces visually pleasing results. Once the map has been constructed, the motion volume is rendered with transparency to produce a motion-blurred image. Figure 2.12 illustrates the process.

Assessment

This algorithm is well-suited to low quality interactive applications, but it suffers from a number of deficiencies that prevent it from producing high quality results. One major issue is that it fails to solve the visibility problem. If two spheres, one in front of the other, are moving perpendicular to the camera plane, only the front sphere should be visible. With this algorithm, both objects will be visible. A second limitation is that the algorithm can only handle polyhedra of genus 0, as these are the only objects that can be separated into a "front" and a "back" as required by the algorithm. This front/back distinction also causes the algorithm problems in the case of rotational motion.

2.7 Conclusion

Although the motion blur problem has been attacked from many different angles over the years, none of the resulting algorithms is entirely satisfactory. In terms of quality, correctness, and generality, the most promising approach is the point–sampling Distribution Ray Tracing algorithm, while various image-based techniques are most often used when speed, not correctness, is of the essence.

In this thesis, we are concerned with computing correctly motion–blurred images for scenes containing arbitrarily complex object motions and appearances. Currently, Distribution Ray Tracing is the only algorithm capable of handling such scenes. Unfortunately, Distribution Ray Tracing is computationally expensive, as it requires extremely high sample counts to avoid noise and banding artifacts in the resulting images. In the rest of this thesis, we examine the causes of these artifacts, and devise new point–sampling strategies for producing high–quality motion–blurred images using fewer point samples than Distribution Ray Tracing.

Chapter 3

Temporal Visibility

3.1 Introduction

If we ignore the question of spatial antialiasing, the motion blur integral reduces to the formulation used by Korein and Badler [8]:

$$I_{xy} = \sum_{i=1}^{N} \int_{0}^{T} h_t(t_{mid} - t) G_i(x_c, y_c, t) I_i(x_c, y_c, t)\, dt \tag{3.1}$$

In this formulation, the final colour of a pixel is computed by temporally filtering the intensity seen through the pixel centre (x_c, y_c) over the exposure time interval. Each object's contribution to the integrand is the product of a visibility term $G_i(x_c, y_c, t)$ that specifies when the object is visible through the pixel centre, and an intensity term $I_i(x_c, y_c, t)$, which specifies the object's colour when visible. In this chapter, we will refer to these functions simply as $G_i(t)$ and $I_i(t)$, with the implicit understanding that they are always evaluated at the pixel centre (x_c, y_c). By identifying the temporal visible intervals T_i during which each object is visible, we can rewrite (3.1) as:

$$I_{xy} = \sum_{i=1}^{N} \int_{T_i} h_t(t_{mid} - t) I_i(t)\, dt \tag{3.2}$$

This formulation suggests a two-step approach to computing the final pixel colour. First, the visible intervals T_i must be computed, then we must compute the integrals of the intensity functions over these visible intervals (we will refer to these as the *shading integrals*). In this chapter, we address the first step, computation of the visible intervals. Chapter 4 will discuss algorithms for computing the shading integrals.

In order to focus on the visible interval computation, we will assume in this chapter that all objects are constant-coloured, that is, $I_i(t) = C_i$ for some constant colour C_i. By making this assumption, we guarantee that any artifacts in the resulting images will be directly attributable to the visible interval computation. Under this assumption, (3.2) reduces to:

$$I_{xy} = \sum_{i=1}^{N} C_i \left(\int_{T_i} h_t(t_{mid} - t)\, dt \right) \tag{3.3}$$

Several previous algorithms have been developed for computing visible intervals for motion blur. Analytic formulations such as those of Korein and Badler [8] and Grant [6] compute the visible intervals exactly, but only for restricted classes of objects (polygons and spheres) moving uniformly in screenspace. Point-sampling algorithms such as Distribution Ray Tracing [4] are capable of handling a wider range of geometric primitives and more complex motions, but they can only approximate the visible intervals, and the resulting images suffer from excessive noise and banding artifacts at low sampling densities.

In this chapter, we will develop a new point-sampling algorithm for approximating the visible intervals in a way that will avoid the artifacts of previous approaches. In Section 2, we describe how the endpoints of the visible intervals can be seen as discontinuities in the temporal "visibility function" for the pixel, and discuss generally how point samples can be used to estimate the locations of these discontinuities. In Section 3, we examine several point-sampling algorithms for locating a single temporal visibility discontinuity, and show that the Random Seed Bisection (RSB) algorithm developed by Ouellette and Fiume [11] is superior to other approaches. In Section 4, we show how Ouellette and Fiume's Two Discontinuity Finding (TDF) heuristic can be used in conjunction with RSB to handle at most 2 visibility discontinuities. We then state our conclusions in Section 5.

3.2 Approximating Visible Intervals by Point Sampling

The visible interval T_i for an object is defined in terms of its visiblility function $G_i(t)$ by the relation $t \epsilon T_i$ iff $G_i(t) = 1$. Thus, the endpoints of the visible interval T_i are located either at the endpoints of the exposure time interval $[0, T]$, or at points where the visibility function is discontinuous. Figure 3.1 illustrates the visibility function and corresponding visible interval for a single moving object.

If we ignore partial visibility due to transparency, then only one object is visible through a given pixel

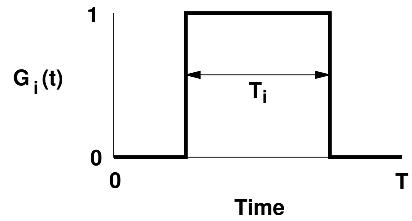


Figure 3.1: Visibility function G_i and visible interval T_i for a single object.

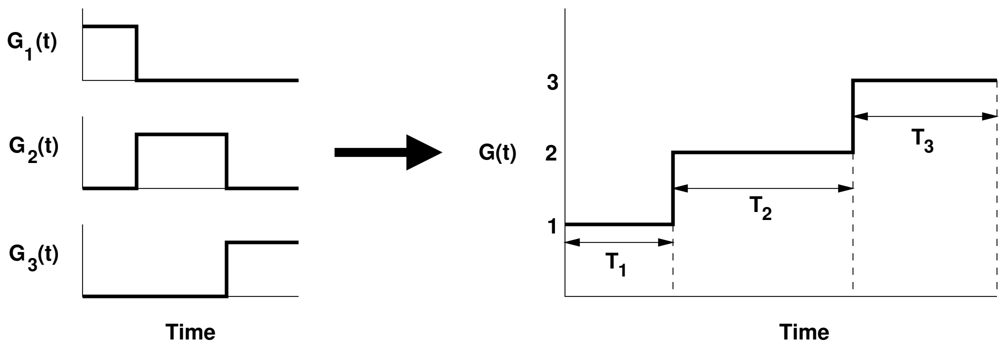


Figure 3.2: Individual object visibility functions G_1, G_2, G_3 and the corresponding global visibility function G.

centre at a given time, so the visibility functions $\{G_i : i = 1 \ldots N\}$ are orthogonal in the sense that

$$G_j(t) = 1 \Rightarrow G_i(t) = 0 \quad \forall\, i \neq j \tag{3.4}$$

We can approximate the locations of the visibility discontinuities by point sampling the visibility functions in time. In a ray tracing system, we can sample visibility at a time t by moving all objects to their positions at time t, casting a *visibility ray* through the pixel centre, and determining the object j that is first hit by the ray. Due to the orthogonal nature of the visibility functions, we can conclude from this that $G_j(t) = 1$ and $G_i(t) = 0 \;\; \forall\, i \neq j$. Thus, we are simultaneously sampling the visibility functions of all the objects in the scene.

Rather than keeping track of N separate but dependent visibility functions, it is more convenient to think of this operation as sampling a single integer-valued *global visibility function* $G(t)$, defined by

$$G(t) = i \quad \text{iff} \quad G_i(t) = 1 \tag{3.5}$$

Figure 3.2 shows the individual visibility functions for several objects, and the corresponding global visibility function.

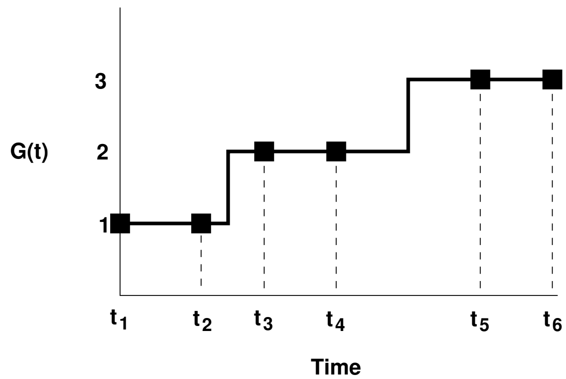


Figure 3.3: Point-sampled approximation of the global visibility function for the example from Figure 3.2. The location of each discontinuity is approximated by the midpoint of the smallest interval known to contain it.

Discontinuities in the global visibility function denote the endpoints of the visible intervals for the pixel. We can approximate the locations of these discontinuities by point sampling the function. Given a set of sample positions $\{t_1, \ldots, t_n\}$ in time, we can conclude that there is a discontinuity in $G(t)$ between any two sample positions t_i and t_{i+1} for which $G(t_i) \neq G(t_{i+1})$. We can approximate the exact location of the discontinuity by the midpoint of the interval $[t_i, t_{i+1}]$, or by a randomly selected location in the interval. Figure 3.3 shows an approximation to the global visibility function from figure 3.2, constructed using 6 visibility samples, with discontinuities approximated at the midpoints of $[t_2, t_3]$ and $[t_4, t_5]$.

Obviously, the quality of the resulting approximation is highly dependent on the *sampling strategy* used to select the sample locations t_i. In the following sections, we will develop sampling strategies for efficiently and accurately approximating the locations of visibility discontinuities. From these approximations, the visibility function can be reconstructed, and images generated.

3.3 Sampling Strategies for Locating a Single Discontinuity

In this section, we consider sampling strategies for approximating the location of a single temporal visibility discontinuity. In Section 3.3.1, we construct a simple scene to illustrate how discontinuity times can vary as a function of position in image space. Section 3.3.2 describes the criteria that we will use in evaluating our approximations of these discontinuity locations. Using these criteria, we then evaluate a set of *adaptive sampling strategies* (in Section 3.3.3), described by Ouellette and Fiume [11], as well as a set of *non-adaptive sampling strategies* (in Section 3.3.4) such as Stratified Random Sampling. We examine some higher-quality results produced by each of the algorithms in Section 3.3.5, then present our conclusion in Section 3.3.6 that of all the strategies presented, Ouellette and Fiume's Random Seed Bisection produces the best results.

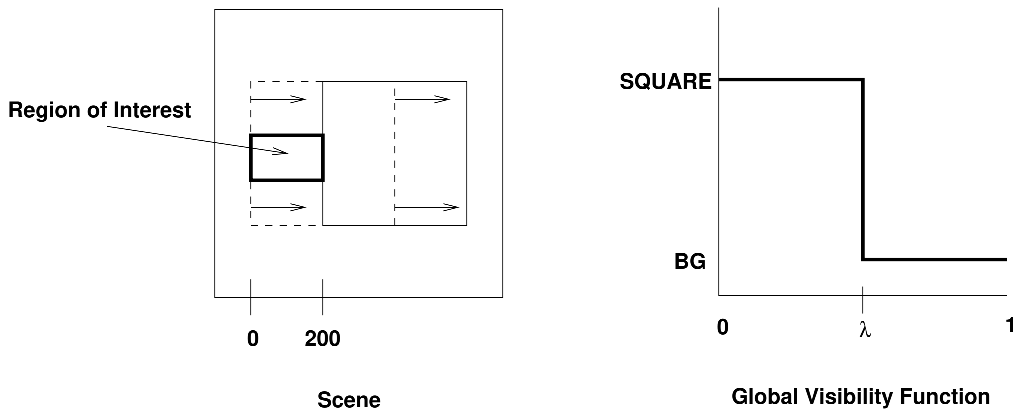
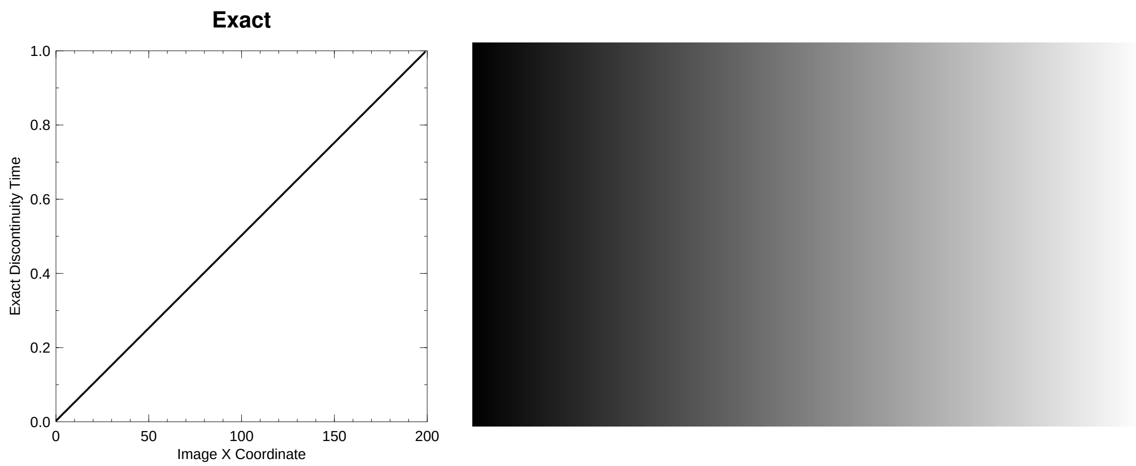


Figure 3.4: Example of a scene containing pixels having exactly one visibility discontinuity. The diagram on the left depicts the scene, consisting of a square moving horizontally from the dotted initial position to the solid final position. All pixels in the bold rectangular region of interest contain exactly one visibility discontinuity, caused by the left edge of the square as it moves from $x = 0$ to $x = 200$ over the exposure time interval $[0, 1]$. The figure on the right illustrates the general form of the global visibility function in these pixels, where the discontinuity occurs at time λ.

3.3.1 Single Discontinuity Example

We will evaluate the algorithms presented in this section using a simple test case consisting of a square moving horizontally to the right, as depicted in figure 3.4. We will concentrate our attention on the rectangular region shown in bold in the image. At the start of the exposure time interval, all pixels in this region are covered by the square. As the square moves right, it gradually uncovers pixels in the region. Since each pixel is uncovered exactly once, we are guaranteed that each pixel centre in the region experiences exactly one discontinuity in its global visibility function over the exposure time interval. The global visibility function for each pixel centre in the region is of the general form shown in the right half of Figure 3.4.

The discontinuity time λ for a particular pixel depends on the location of the pixel in the region, with pixels on the left having earlier discontinuity times. Figure 3.5 shows the exact discontinuity times plotted as a function of horizontal position in the region. Note that the pixel discontinuity times vary continuously as a function of position along the image-space trajectory of the object (a property that holds for any continuous object motion, even non-linear motion). Using Equation 3.3 to compute a motion-blurred image (shown at right in Figure 3.5) of the region, we see that the result of this smooth variation of discontinuity times results in a smooth colour variation across the image, from the background colour (black) at left to the object colour (white) at right. This image is "ideal" in that it was computed using discontinuity times that were computed exactly by analytic intersection with the moving object. We assume that the temporal filter $h_t(t)$ is a box filter, so the exact pixel colour is simply the width of the object's visible interval. This image



Exact Discontinuity Time $\lambda(x)$ **Exact Motion-Blurred Image**

Figure 3.5: Exact discontinuity times for pixels in the region marked in Figure 3.4, and an image computed from these discontinuity times. The moving object is white against a black background, hence the tail of the moving object fades from white to black, as shown in the image.

is the standard against which we will compare the results of our single–discontinuity finding algorithms. The following section describes the criteria used for making these comparisons.

3.3.2 Evaluation Criteria

Since most of our discontinuity–finding algorithms incorporate some degree of randomness into their search, we can consider each to be a probabilistic estimator $F(x)$ of the actual discontinuity location $\lambda(x)$, where x represents the image-space x-coordinate of the pixel being considered. The expectation $E[F(x)]$ and the variance $V[F(x)]$ are themselves both functions of x as well. By running a series of trials at each pixel centre in the region, we can compute the mean and variance of our estimator, and plot the results on an *expectation/variance graph* like the one shown in figure 3.6. The thick line represents the mean $E[F(x)]$, and the thin lines represent an interval of one standard deviation about the mean (the curves are $E[F(x)] \pm 0.5 \times \sqrt{V(x)}$). Using these plots, we can evaluate an estimator based on the following criteria:

Continuity of Expectation. The expectation $E[F(x)]$ of the estimator should be roughly equal to $\lambda(x)$, since that is the true location of the discontinuity. More importantly, the expectation should be a **continuous** function of x in order to ensure a smoothly blurred image. Any discontinuities in the expectation curve will almost certainly result in objectionable banding artifacts in the resulting image.

Constancy of Variance. The variance of the estimator measures how widely we can expect individual estimates $F(x)$ to deviate from the mean $E[F(x)]$. Variance in the estimator manifests itself as noise

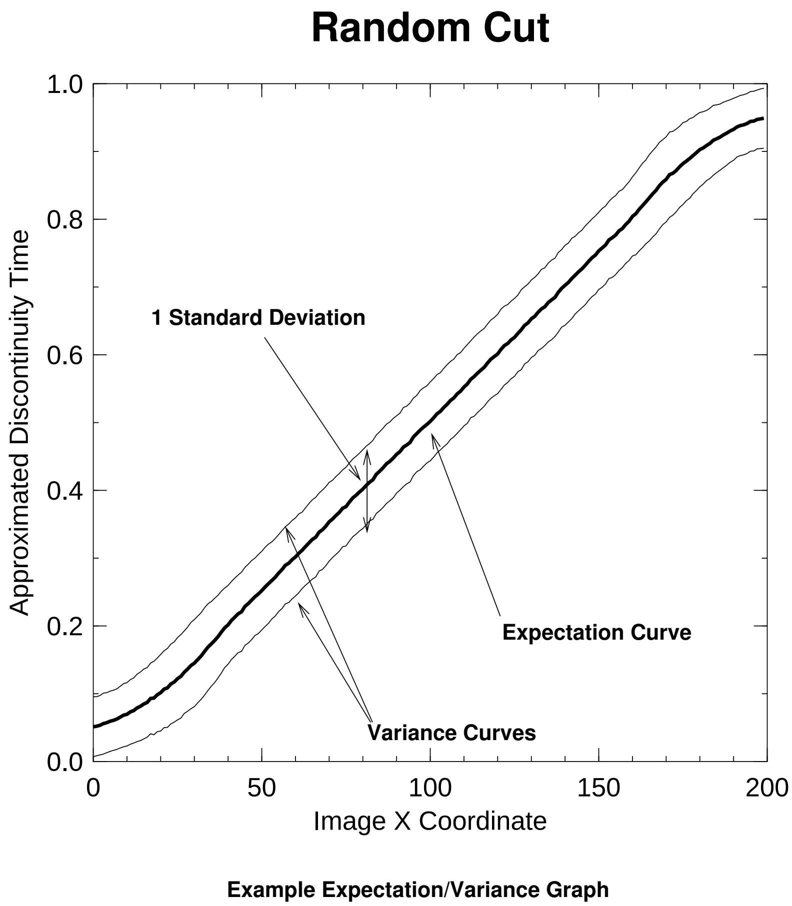


Figure 3.6: Example of an expectation/variance graph. The thick line represents the expectation of the estimator as a function of the image x coordinate. The thin lines represent a distance of one standard deviation about the expectation.

in the image, with high variances producing noisier images. It is important that the magnitude of this noise be uniform over the entire domain of consideration, otherwise objectionable noise artifacts may result. To this end, we desire that the variance be roughly **constant** for all values of x. On the graph in Figure 3.6, variance is expressed as the vertical distance between the light–coloured variance curves.

Convergence Rate. All other things being equal, we would like our estimator to converge to its final result as quickly as possible. We measure convergence rate by the average number of samples per pixel required by the estimator, averaged over all pixels in the region of interest.

3.3.3 Adaptive Sampling Strategies

Adaptive sampling strategies for locating a single visibility discontinuity are essentially just binary search algorithms. Given a time interval $[t_a, t_b]$ known to contain a visiblility discontinuity (i.e. $G(t_a) \neq G(t_b)$), we sample the visibility function at a "cut point" $t_c \in [t_a, t_b]$. We are assuming that the interval contains exactly one visibility discontinuity, so $V(t_c)$ must equal either $V(t_a)$ or $V(t_b)$. If $V(t_c) = V(t_a)$, then the discontinuity must lie in the interval $[t_c, t_b]$. Otherwise, the discontinuity must lie in $[t_a, t_c]$. We recurse on the subinterval known to contain the discontinuity, discarding the other subinterval. The recursion continues until the width of the interval being searched drops below a specified tolerance ϵ, at which point we make a final cut and return this cut position as our estimate of the true discontinuity location.

As visibility samples are cast, more information about the visibility function is revealed, and the sampling pattern adapts itself to reflect this information by pruning away intervals that could not contain the discontinuity (hence the term *adaptive sampling*). The exact behaviour of the algorithm is determined by the method used to select the cut locations. Each cut can be made deterministically (at the midpoint of the interval), or probabilistically (at a randomly selected position in the interval). Furthermore, different methods may be used for the initial cut, the final cut, and for all other cuts in between. Figure 3.7 lists a code template for adaptive single discontinuity finders, as well as four separate instantiations of the template based on different cut strategies. Expectation/variance plots and sample images for each of the algorithms (using a rather high tolerance $\epsilon = 0.2$) are shown in Figures 3.8 and 3.9. We will discuss each in turn:

Pure Bisection (PB)

When all cuts are made at the midpoint of the interval, the result is the Pure Bisection algorithm. It is clear from the graph that this algorithm fails the continuity of expectation criterion. Instead of a smooth variation, the expectation curve exhibits a stairstep behaviour. As a result, "banding artifacts" are present

```
bool findOneDisc_Adaptive( float a, float b, float& disc )
{
    if( G(a) == G(b) ) {
        return false;
    }

    if( (b - a) < ε ) {
        disc = finalcut(a,b);
        return true;
    } else {
        float c = cut(a,b);
        if( G(c) == G(a) ) {
            return findOneDisc_Adaptive(c,b,disc);
        } else {
            return findOneDisc_Adaptive(a,c,disc);
        }
    }
}
```

Pure Bisection:
cut(a,b) = $(a+b)/2$
finalcut(a,b) = $(a+b)/2$

Jittered Bisection:
cut(a,b) = $(a+b)/2$
finalcut(a,b) = random(a,b)

Random Cut:
cut(a,b) = random(a,b)
finalcut(a,b) = random(a,b)

Random Seed Bisection:
cut(a,b) = random(a,b) (first time)
cut(a,b) = $(a+b)/2$ (subsequently)

Figure 3.7: Adaptive single discontinuity finding algorithms. The "findOneDisc_Adaptive" function attempts to locate a visibility discontinuity in the time interval $[a, b]$, returning true if one was found, and false otherwise.

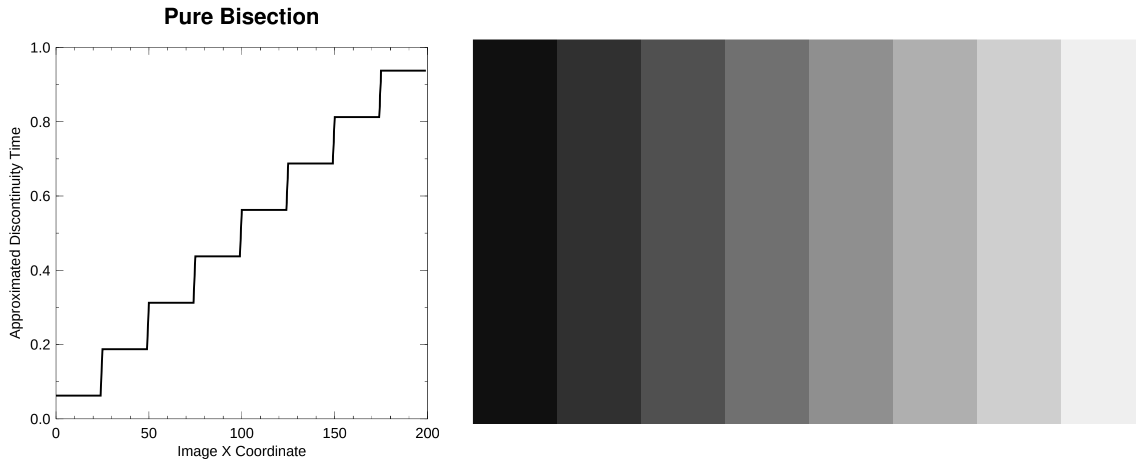
in the resulting image. Instead of the desired smooth colour variation across the image, we see a series of distinct bands of constant colour.

Despite this drawback, the Pure Bisection algorithm satisfies our constancy of variance criterion. Since it is completely deterministic, the variance is trivially zero (hence the absence of variance curves on the graph). Pure Bisection also provides the (provably) optimal covergence rate, requiring an average of 5 samples per pixel for our error tolerance of $\epsilon = 0.2$.

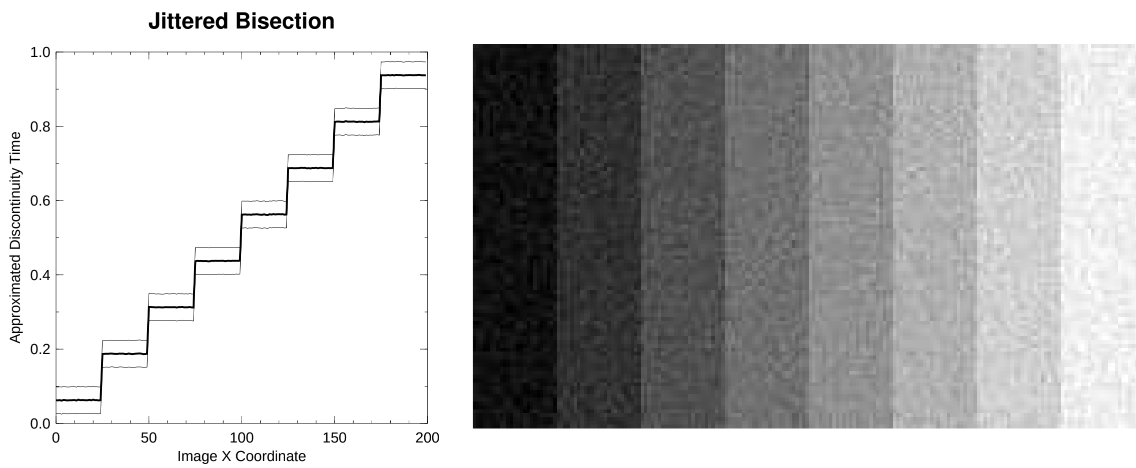
Jittered Bisection (JB)

To alleviate the banding artifacts of Pure Bisection, we introduce some randomness into the algorithm. If we make the final cut random (with all other cuts at interval midpoints), the result is the Jittered Bisection algorithm. The algorithm satisfies the constancy of variance criterion (as shown by the roughly constant vertical distance between the variance curves in the graph), so the amount of noise is roughly constant over the entire image. Furthermore, Jittered Bisection converges just as quickly as Pure Bisection, requiring an average of 5 samples per pixel.

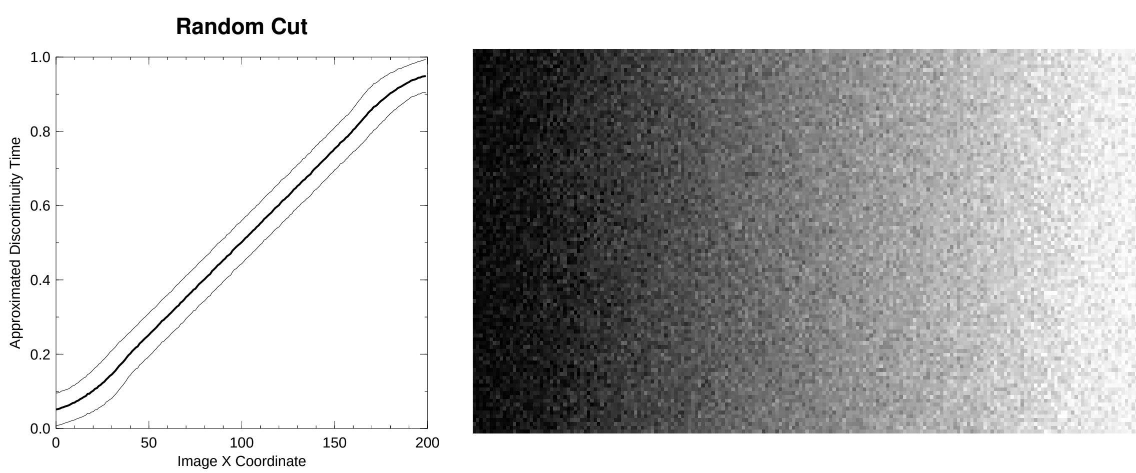
Unfortunately, Jittered Bisection fails the continuity of expectation criterion, as demonstrated by the familiar stairstep nature of the expectation curve. As a result, the JB image is simply the PB image with



Pure Bisection: 5.0 samples/pixel

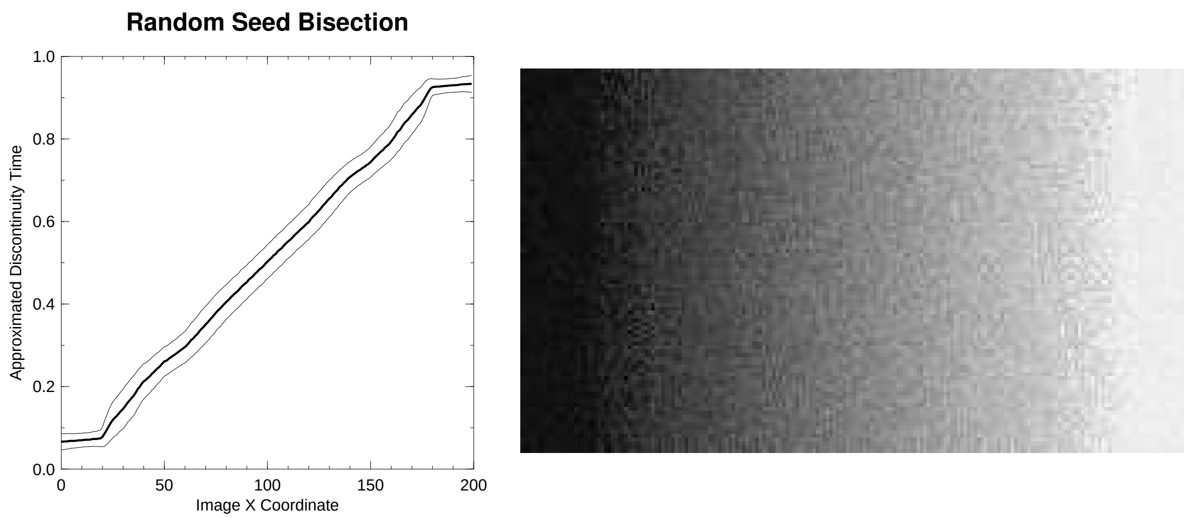


Jittered Bisection: 5.0 samples/pixel

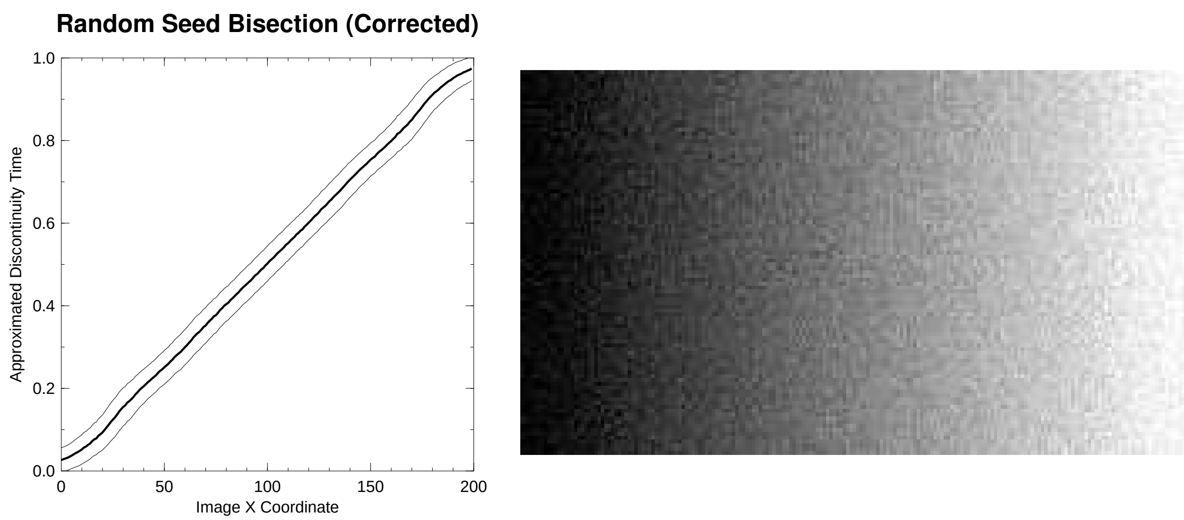


Random Cut: 6.21 samples/pixel

Figure 3.8: Adaptive algorithm results ($\epsilon = 0.2$): Expectation/variance plots for adaptive single discontinuity finders applied to the test case. The average number of visibility samples per pixel is given for each algorithm.



Uncorrected Random Seed Bisection: 5.16 samples/pixel



Corrected Random Seed Bisection: 5.71 samples/pixel

Figure 3.9: Adaptive algorithm results ($\epsilon = 0.2$), cont'd: Expectation/variance plots and motion-blurred images for the Random Seed Bisection algorithm applied to the test case. Use of the corrected algorithm eliminates the end bands present in the uncorrected version, at a cost of an extra 0.55 samples per pixel.

some added noise. Instead of bands of constant colour, JB produces bands of noisy colour.

Random Cut (RC)

By making all cuts probabilistic, we obtain the Random Cut algorithm. The graph shows that this algorithm satisfies the continuity of expectation criterion, resulting in an image free of banding artifacts. The image is rather noisy, but at least the amount of noise appears to be constant over the entire image. The graph confirms that although the variance is high, the constancy of variance criterion is satisfied. The convergence rate is slower than PB and JB, with RC requiring 6.21 samples per pixel.

Random Seed Bisection (RSB)

Random Seed Bisection attempts to retain the continuity of expectation of RC while avoiding its slow convergence and high variance. RSB makes the first cut probabilistically, and all subsequent cuts deterministically. The result is that between $x = 20$ and $x = 180$, the expectation curve is continuous, the variance is roughly constant and lower than the variance of RC, and the convergence (5.16 samples/pixel) is nearly optimal. It would appear that over this range, RSB produces the best results of all the algorithms.

Unfortunately, RSB exhibits quite different behaviour when $x < 20$ and when $x > 180$, where the expectation curve flattens out. The result is that the smoothly blurred middle region of the image is surrounded by two “end bands”, a dark band on the left and a light band on the right. The effect is not as visually disturbing as the pervasive banding artifacts of PB or JB, but it is clearly unacceptable.

The cause of problem is that all discontinuities that lie within $\frac{\epsilon}{2}$ of the endpoints of the exposure time interval are essentially indistinguishable to the algorithm, since samples are almost never cast there. The only way that a non-random cut could occur so close to the endpoint would be as a result of cutting an interval of width less than ϵ, which never happens. It is possible for an initial random sample to be cast that close to the endpoint, but this happens infrequently.

Thus, the width of the bands is a function of the error tolerance ϵ. As ϵ is lowered, the bands become narrower, until they are barely noticeable at $\epsilon < 0.05$. It is common to use error tolerances this low when generating high-quality images, so it is possible that the problem will just solve itself. If low quality ($\epsilon \geq 0.2$) images are being generated, we can solve the problem by simply expanding the range being searched. Instead of searching for the discontinuity in the time interval $[0, 1]$, we instead search $[-\frac{\epsilon}{2}, 1 + \frac{\epsilon}{2}]$, and clamp our computed discontinuities to the interval $[0, 1]$. The result of this “Corrected RSB” is shown in Figure 3.9. The effect of this change is to simply force the end bands outside of the exposure time interval, leaving us with a continuous–expectation, constant–variance estimator that converges faster and has lower variance than RC. There is a price to be paid, in that about 9 percent more samples are required than for uncorrected RSB, but the convergence is still faster than RC.

Thus, it appears that based on our criteria, RSB is the best adaptive sampling algorithm for locating a single discontinuity. The next section explores several non–adaptive sampling strategies, and compares them to our adaptive results.

3.3.4 Stratified Random Sampling and Other Non-Adaptive Sampling Strategies

In the adaptive algorithms described in the previous section, each sample position is dependent on the visibility computed at all the previous sample positions. By contrast, *non-adaptive* algorithms precompute the entire set of sample positions before any visibility samples are taken. We will examine two such algorithms, Uniform Sampling (US) and Stratified Random Sampling (SRS). Stratified Random Sampling is the sampling strategy at the heart of Distribution Ray Tracing, currently the most widely used point–sampling motion blur algorithm.

Both algorithms work by subdividing the exposure time interval into n strata and casting a single visibility sample t_i in each stratum. Uniform sampling casts samples at the centre of each stratum, while Stratified Random Sampling casts samples at randomly determined positions within each stratum. It is assumed that visibility is constant over each stratum, so the global visibility function can be reconstructed as a step function, i.e. a sum of constant-valued basis functions. More formally,

$$G(t) = \sum_{i=0}^{n-1} G(t_i) B_i(t) \tag{3.6}$$

where

$$B_i = \begin{cases} 1 & \text{if } t \,\epsilon\, [\frac{i}{n}, \frac{(i+1)}{n}] \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

$$t_i = \begin{cases} \frac{(i+0.5)}{n} & \text{for Uniform Sampling} \\ \frac{(i+random(0,1))}{n} & \text{for Stratified Random Sampling} \end{cases} \tag{3.8}$$

Figure 3.10 shows an example of a visibility function reconstructed from US and SRS sampling patterns. As was the case with the adaptive algorithms, we can consider these non-adaptive algorithms as estimators of the location of visibility discontinuities (since the reconstructed visibility function is discontinuous). Figure

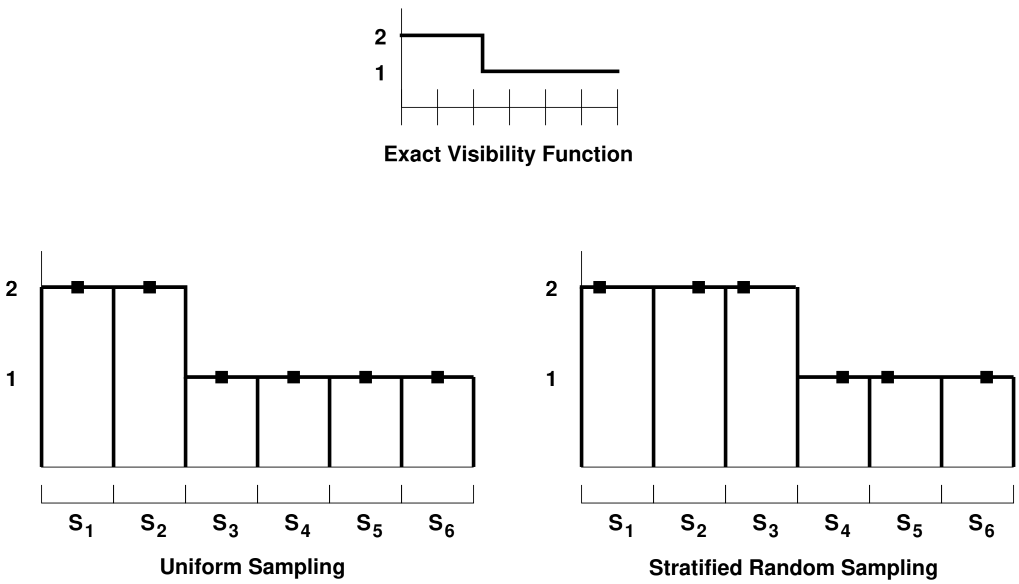


Figure 3.10: Reconstructing the visibility function as a sum of constant-valued basis functions. One sample is cast in each stratum S_i and used to weight the basis function B_i for that stratum. For Uniform Sampling, the reconstructed function is $2B_1 + 2B_2 + 1B_3 + 1B_4 + 1B_5 + 1B_6$, and for Stratified Random Sampling, it is $2B_1 + 2B_2 + 2B_3 + 1B_4 + 1B_5 + 1B_6$.

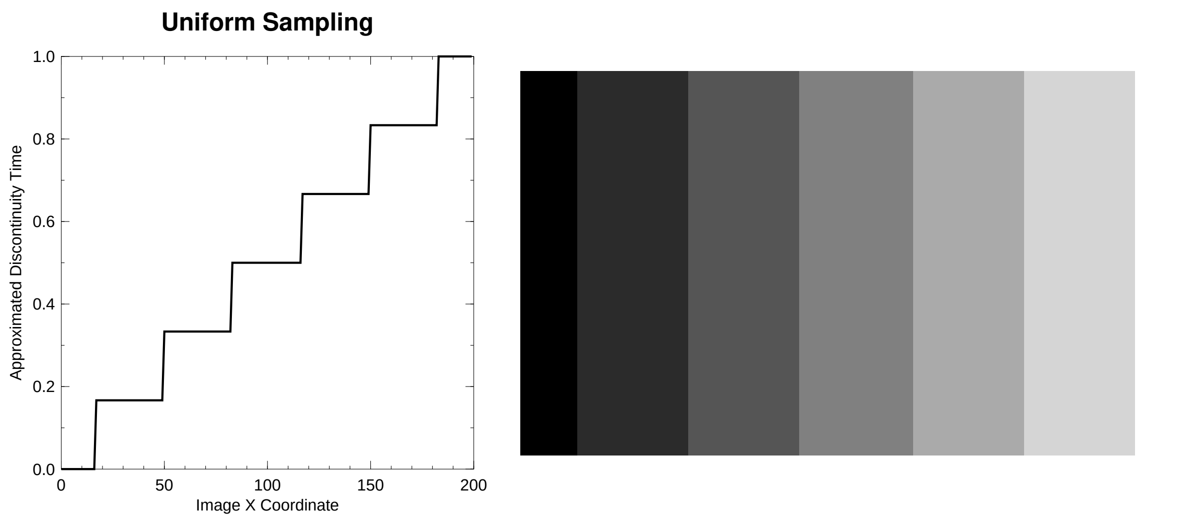
3.11 gives the expectation/variance plots and sample images for both of these estimators. Six strata (and hence 6 visibility samples) are used in each case, as this was approximately the number of samples used by the adaptive algorithms.

Uniform Sampling (US)

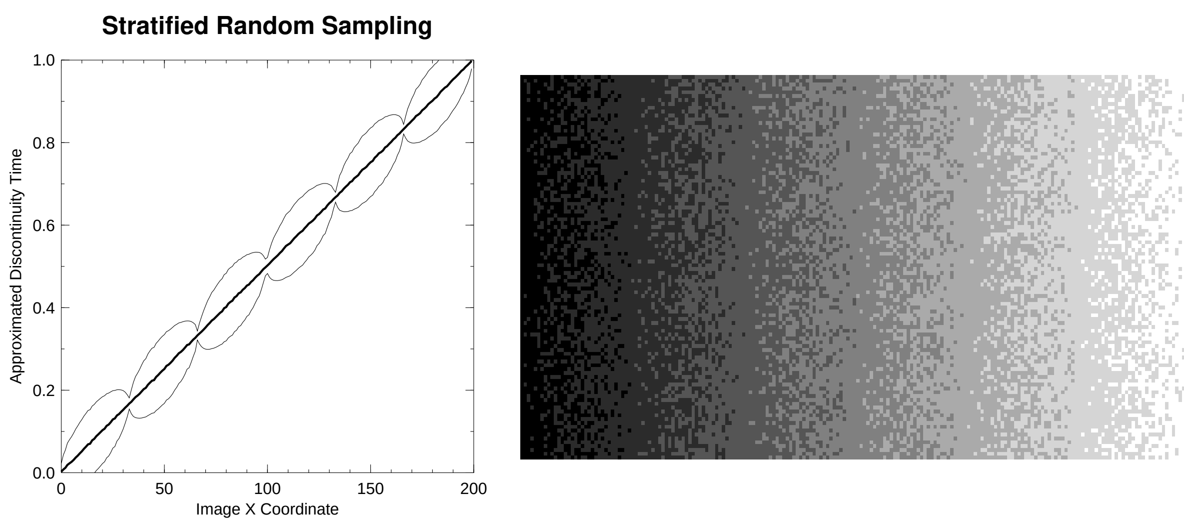
The graph for this deterministic algorithm exhibits the familiar stairstep expectation curve that we observed with PB in the previous section, and the accompanying constant-coloured banding artifacts in the image. The bands produced by US are wider (and hence more objectionable) than those of PB due to the fact that US tends to cast many of its samples far away from the actual discontinuity location.

Stratified Random Sampling (SRS)

The SRS image exhibits noticeable banding artifacts, despite the fact that it satisfies the continuity of expectation requirement, as shown in the graph. The source of the banding in this case is the variance behaviour of the estimator. The graph clearly shows that the constancy of variance criterion is not satisfied. The estimator oscillates between areas of high variance in the middle of each stratum, and areas of almost no variance near the endpoints of the strata. To see why this is the case, consider that if a discontinuity lies close to a stratum endpoint, most random samples cast in that stratum will place the estimated discontinuity



Uniform Sampling



Stratified Random Sampling

Figure 3.11: Non-adaptive algorithm results (6 samples/pixel): Expectation/variance plots and motion-blurred images for the Uniform Sampling and Stratified Random Sampling non-adaptive single discontinuity finding algorithms. Both algorithms were run using 6 samples per pixel.

at that endpoint, resulting in a low variance estimator. In fact, the variance is exactly zero when the discontinuity lies exactly on a stratum endpoint. If the discontinuity lies in the middle of the stratum, however, both stratum endpoints have an equal chance of being chosen as a discontinuity estimate, resulting in a higher variance estimator.

The result of this non-constant variance behaviour is that instead of a smooth blur, the image consists of bands of noise separated by bands of roughly constant colour. This banding is less objectionable than that of PB, JB, and US, but the result is clearly inferior to that of RC and RSB.

3.3.5 Higher Resolution Results

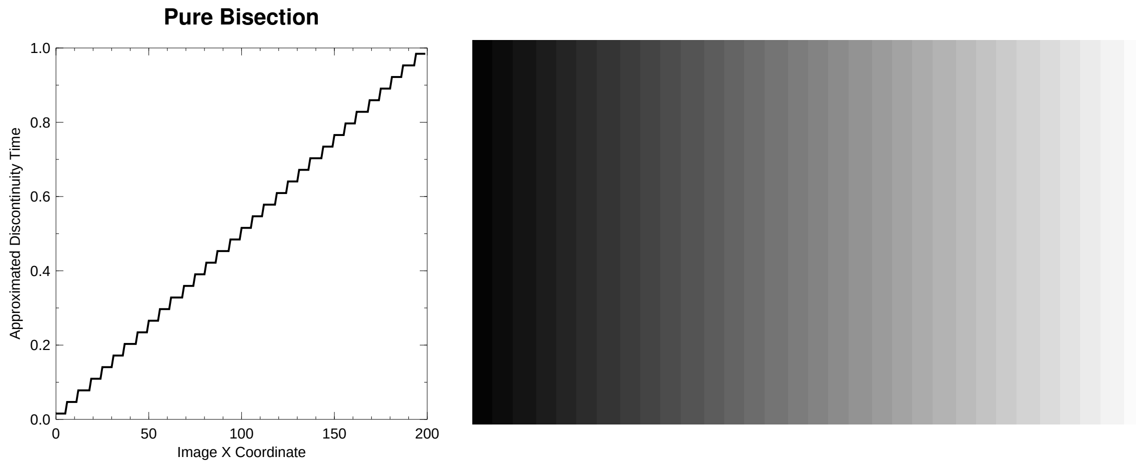
The images and analyses presented in the preceding sections were based on applying the algorithms at low sample counts, in order to accentuate the differences between them. The non-adaptive algorithms used exactly 6 samples per pixel, and the adaptive algorithms used an error tolerance of $\epsilon = 0.2$, which required between 5 and 6 samples on average. Figures 3.12, 3.13, and 3.14 show some higher-quality images produced using an error tolerance of 0.05 for the adaptive algorithm, and 8 samples per pixel for the non-adaptive algorithms.

At this lower error tolerance, the properties of the algorithms remain the same, but the differences are more subtle. PB and JB still produce banding artifacts, but the bands are much narrower and less noticeable. RC produces a smooth image, but slightly noisier and with slower convergence (9 samples/pixel) than RSB (7.15 samples/pixel). RSB's end bands are still present, but they are so narrow as to be almost unnoticeable. The need to correct the algorithm becomes debatable at this point, although the correction costs only an extra 0.2 samples/pixel on average.

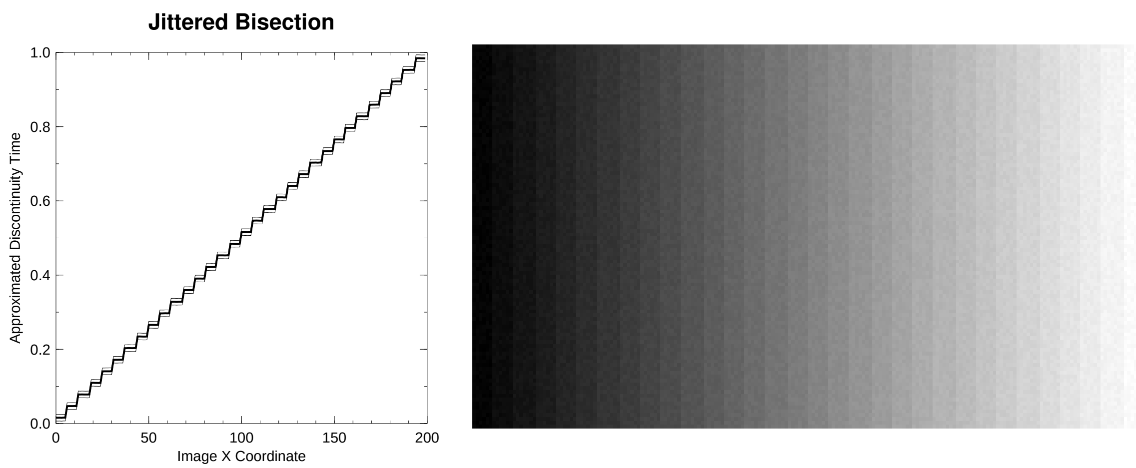
At this quality level, the differences between adaptive and non-adaptive algorithms become even more evident. To illustrate the extent of the problem, results are shown in Figure 3.14 for SRS using 8, 20 and 40 samples per pixel. Even the 20-sample result is clearly noisier than the 7-sample RSB result. Only at 40 samples per pixel does the SRS image appear as smooth as the 7-sample RSB.

3.3.6 Conclusions

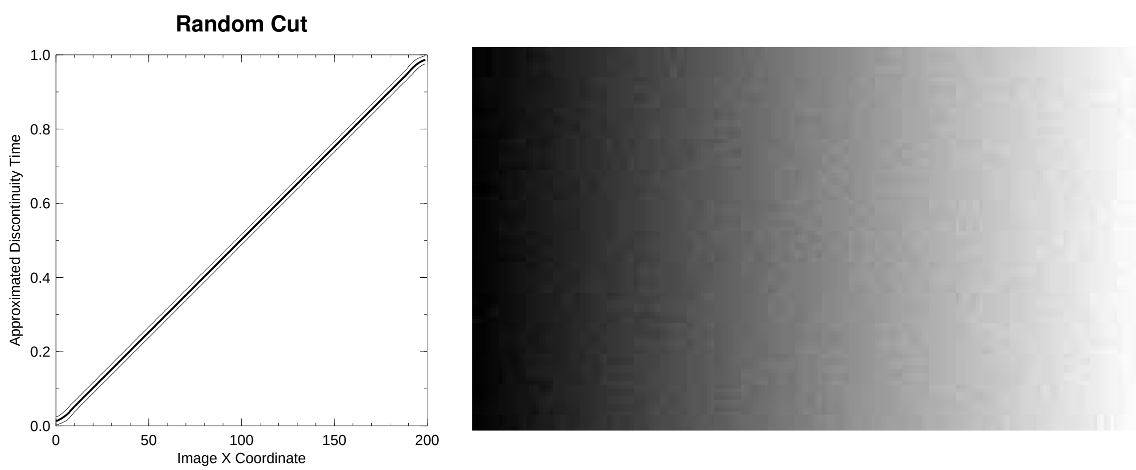
The results presented in this section show that in the presence of a single discontinuity, Random Seed Bisection produces the best results of all the adaptive algorithms, converging quickly to a smooth solution free of banding artifacts. Stratified Random Sampling is the best non-adaptive algorithm, but it suffers from noisy banding artifacts, and its results are clearly inferior to those of RSB at similar sample counts in the presence of a single discontinuity.



Pure Bisection: 7.0 samples/pixel

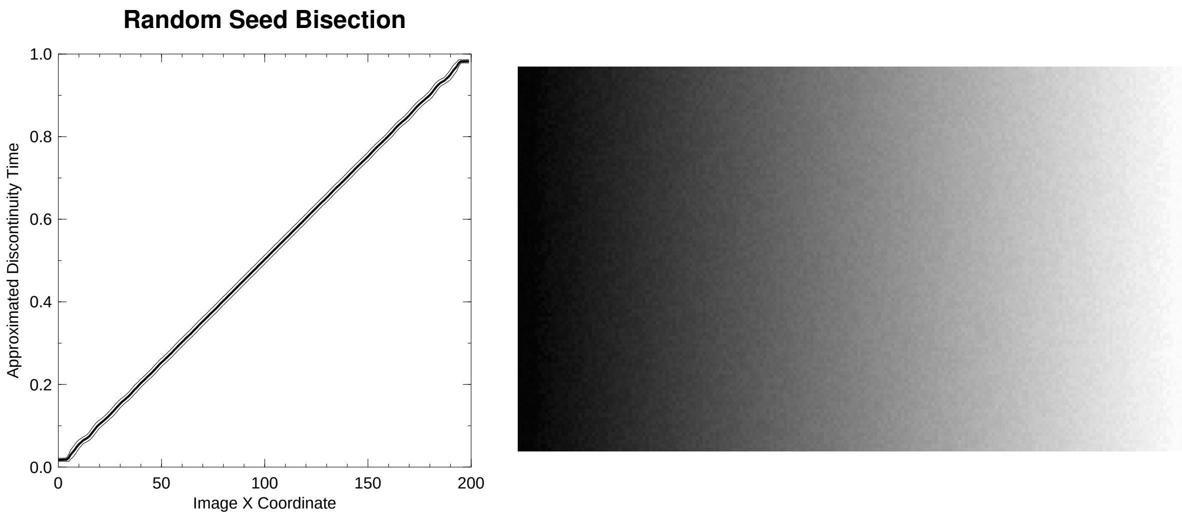


Jittered Bisection: 7.0 samples/pixel

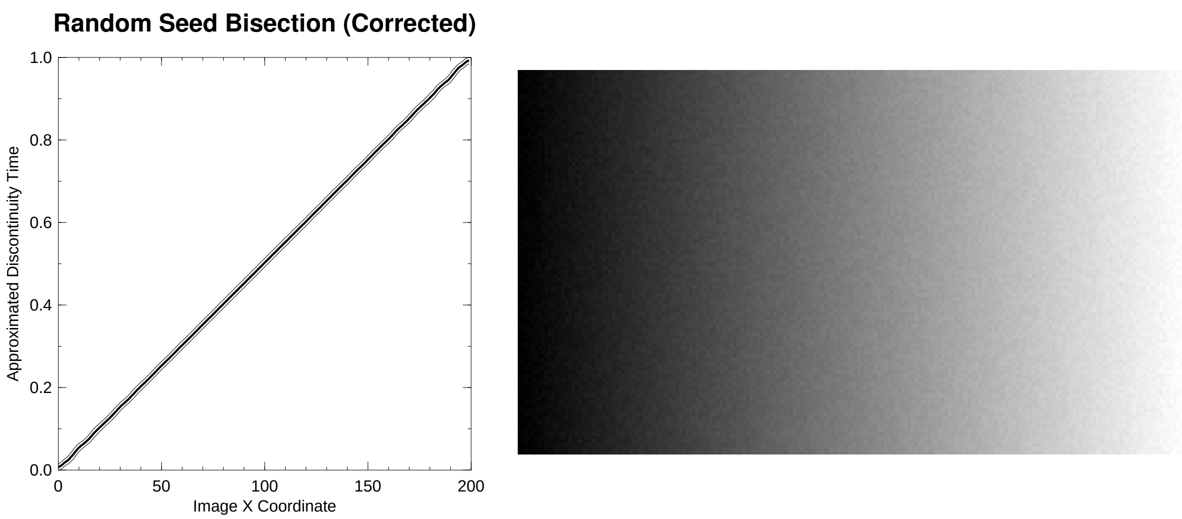


Random Cut: 8.99 samples/pixel

Figure 3.12: High-quality adaptive algorithm results ($\epsilon = 0.05$).

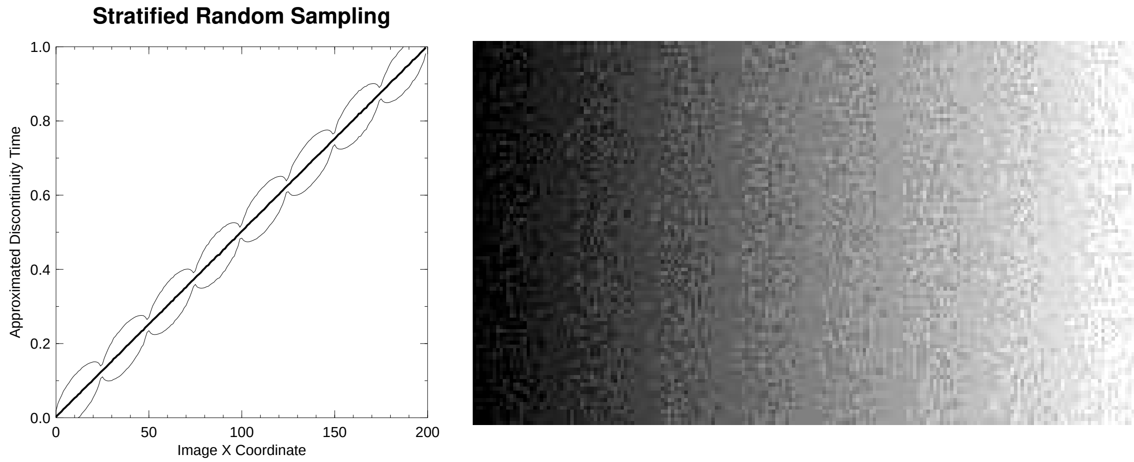


Uncorrected Random Seed Bisection: 7.15 samples/pixel

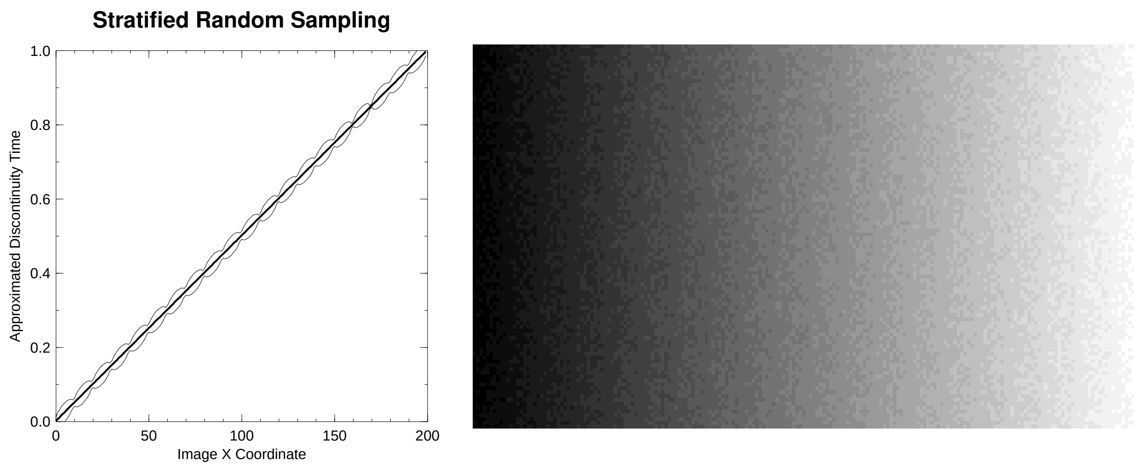


Corrected Random Seed Bisection: 7.35 samples/pixel

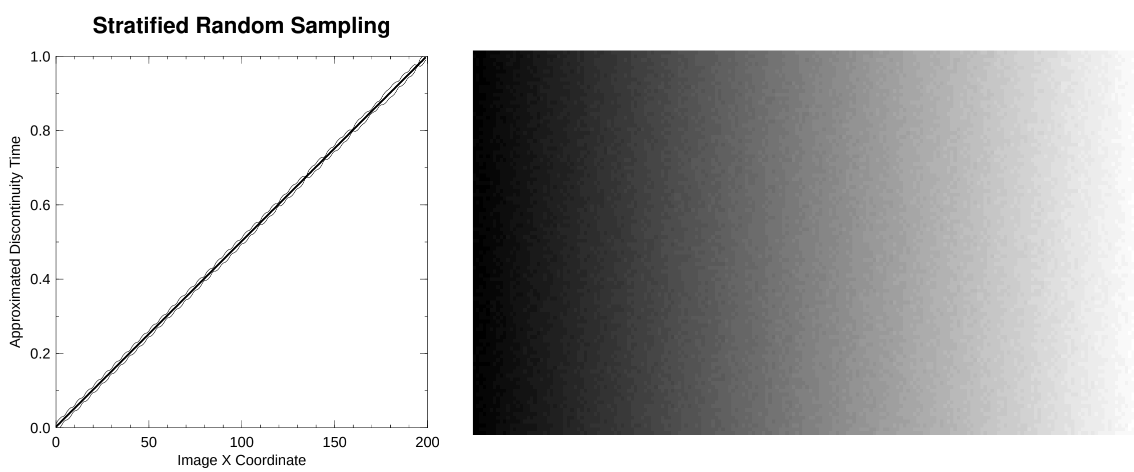
Figure 3.13: High-quality adaptive algorithm results ($\epsilon = 0.05$), cont'd.



Stratified Random Sampling (8 samples/pixel)



Stratified Random Sampling (20 samples/pixel)



Stratified Random Sampling (40 samples/pixel)

Figure 3.14: High-quality non-adaptive algorithm results.

Stratified Random Sampling cannot, however, be discarded based solely on the results presented so far. It is important to note that SRS makes no assumptions about the number of discontinuities in the visibility function. It is fairly common for a pixel to contain two or more visibility discontinuities. The SRS algorithm requires no modification to handle these cases, and given enough samples, will converge to a smooth result. In such a case, RSB will produce a good approximation of one of the discontinuities at best, and at worst it will fail to detect any discontinuities at all. In either case, the result produced by RSB is not immediately useful for reconstructing the visibility function.

It is not obvious how to extend the RSB algorithm to handle multiple discontinuities, but it can be done. The following section describes Ouellette and Fiume's Two-Discontinuity Finding (TDF) heuristic for applying RSB to the case of at most two visibility discontinuities. Using this heuristic, we can compute smoothly blurred images in this more complicated case using fewer samples than SRS. In Chapter 4, we will describe how to extend this idea to the case of an arbitrary number of discontinuities.

3.4 Sampling Strategies for Locating at Most Two Discontinuities

In this section, we discuss algorithms for approximating the location of up to two visibility discontinuities in a pixel. At first, we will restrict ourselves to the case in which the pixel contains **exactly** 2 discontinuities. In Section 3.4.1, we describe a test case containing such two-discontinuity pixels, and examine the criteria that we will use in evaluating our visibility approximations for these pixels. In Section 3.4.2, we discuss how to apply RSB to the two-discontinuity case. The general approach here will be to first split the exposure time interval into two subintervals, each containing a single discontinuity which can then approximated using RSB. We will evaluate several algorithms for performing this split, and conclude that Ouellette and Fiume's Voronoi Search (VS) algorithm provides the best results.

Armed with our algorithms for dealing with exactly 1 discontinuity (RSB) and exactly 2 discontinuities (VS followed by RSB), we still need to decide which algorithm is appropriate for each individual pixel. Section 3.4.3 examines Ouellette and Fiume's Two-Discontinuity Finding (TDF) heuristic for deciding how many discontinuities (0, 1, or 2) a particular pixel contains, and thus which of our algorithms should be applied to it.

3.4.1 Two Discontinuity Example

In the example used in the previous section, the speed of the moving object was such that at most one visibility discontinuity was produced at each pixel. That is, the object was moving fast enough to either

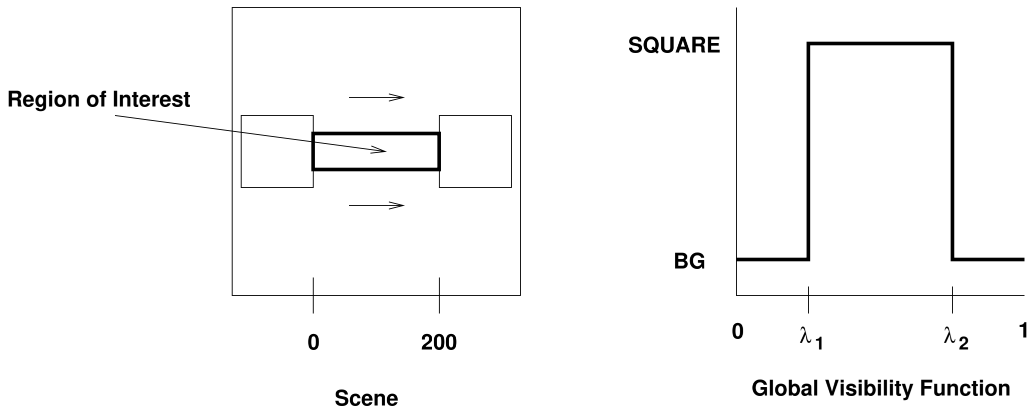


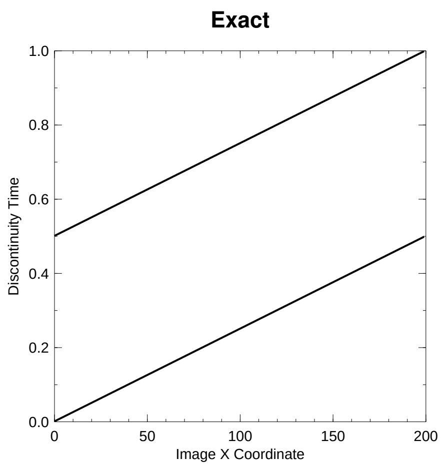
Figure 3.15: Example of a scene containing pixels having exactly two discontinuities. As the square moves right as shown, pixels in the highlighted region experience two visibility discontinuities, first when the square covers the pixel centre at time $t = \lambda_1$, and then when the square uncovers the pixel centre at time $t = \lambda_2$.

enter or leave a particular pixel during the exposure time interval, but not fast enough to pass entirely through the pixel during this time. In this section we consider a faster moving square, one capable of producing 2 visibility discontinuities in a pixel, as shown in Figure 3.15. Each pixel centre in the highlighted region contains exactly two visibility discontinuities, one when the object enters the pixel at time λ_1, and another when the object leaves the pixel at time λ_2. Thus, the global visibility function of each pixel in the region has the general form shown in the figure at right. As in the previous section, the discontinuity times are functions $\lambda_1(x)$ and $\lambda_2(x)$ of image position x, but we will frequently use the shorthand λ_1 and λ_2 to refer to the values of these functions at a particular pixel centre.

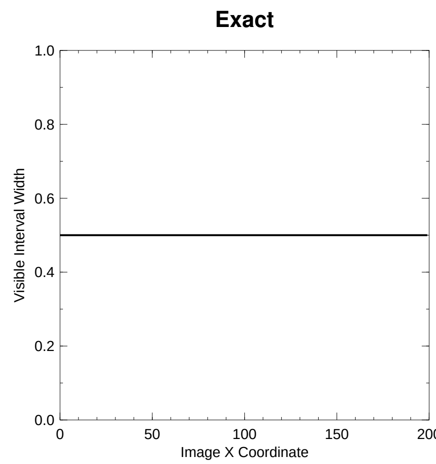
In this example, we again use a white square moving against a black background and a temporal box filter, so when Equation (3.3) is applied to a two-discontinuity pixel, the width $\lambda_2 - \lambda_1$ of the object's visible interval determines the final pixel colour. Figure 3.16 shows a graph of the exact discontinuity times $\lambda_1(x)$ and $\lambda_2(x)$, as well as a graph of the width $W(x) = \lambda_2(x) - \lambda_1(x)$ of the resulting visible intervals. The object moves at a constant speed in our example, so the width $W(x)$ is constant, resulting in the constant-coloured motion-blurred image shown in the figure.

In evaluating our approximation algorithms for this case, we will use similar criteria to those used in the previous section. In particular, we will require that our approximation of each discontinuity location satisfy the continuity of expectation and constancy of variance requirements, but we will also require that the approximated interval width $W(x)$ also satisfy these requirements.

To demonstrate the importance of the width estimate, consider figure 3.17, which shows expectation/variance plots for the estimated discontinuity times and visible interval widths produced by applying 10-sample Stratified Random Sampling to our example. As expected, we observe the characteristic variance banding in the



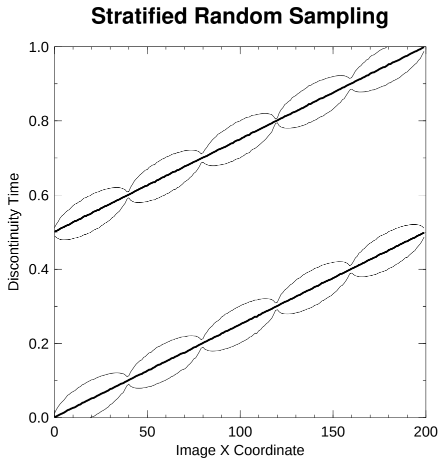
Exact Discontinuity Times $\lambda_1(x)$, $\lambda_2(x)$



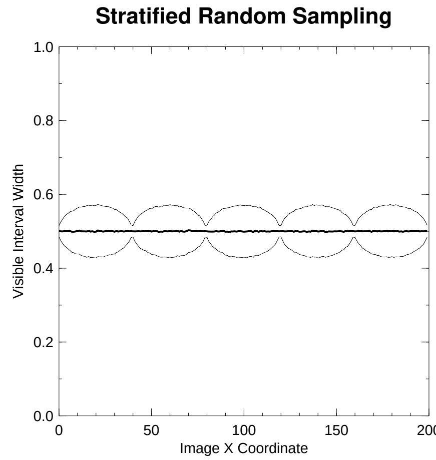
Exact Interval Widths $W(x)$

Exact Motion-Blurred Image

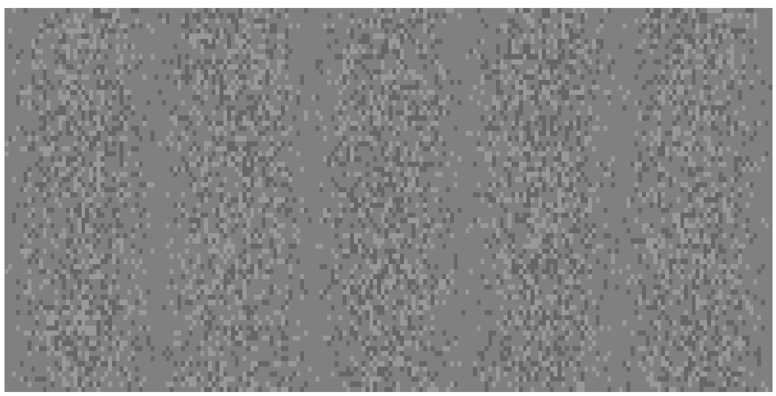
Figure 3.16: Exact discontinuity times and visible interval widths for pixels in the two-discontinuity region of the test scene. The interval widths are used to generate the exact motion-blurred image.



Approximated Discontinuity Times



Approximated Interval Widths



Approximate Motion-Blurred Image

Figure 3.17: Stratified Random Sampling (10 samples): Approximated discontinuity locations, interval widths, and motion blurred image for the two-discontinuity pixels in the example, computed using 10-sample Stratified Random Sampling. The characteristic variance bands of the discontinuity location estimates interact to form variance bands in the width estimate, which are visible in the resulting image.

discontinuity time estimates, but the interaction between these estimates also produces variance banding in the width estimate, as shown in the graph at right. The banding artifacts in the final image are the direct result of the width estimate's violation of the constancy of variance criterion. In the following sections, we describe how to produce better width estimates using RSB.

3.4.2 Splitting the Interval

In order to reap the benefits of RSB in the presence of exactly two discontinuities, we will split the exposure time interval into two subintervals, each containing a single discontinuity. By applying RSB to approximate the discontinuity times in each subinterval, we hope that the resulting width estimates will satisfy the

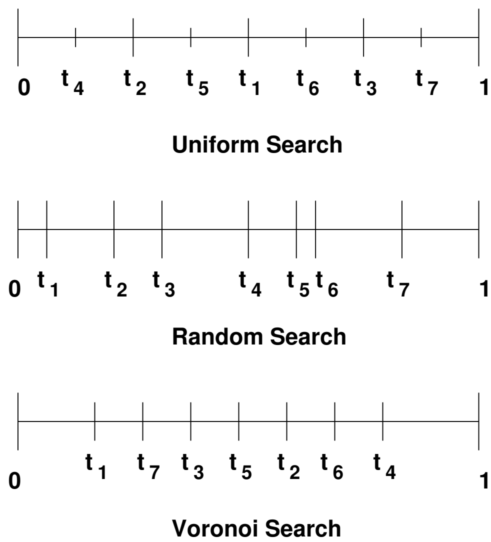


Figure 3.18: Sampling patterns for interval splitting generated by Uniform Search, Random Search, and Voronoi Search.

continuity of expectation and constancy of variance criteria (this seems reasonable, given that each individual discontinuity estimate satisfies these criteria). The problem at hand, then, is how to split the exposure time interval.

If the visibility function is of the general form shown in Figure 3.15, then any time t_s with $V(t_s) =$ SQUARE can be used to split the interval into two halves containing a single discontinuity each. In this section, we will examine three search algorithms for finding such a "split point" t_s. Random Search, Uniform Search, and Voronoi Search each generate a series of times $\{t_i\}$ at which to sample the visibility function, terminating when a suitable split point is found.

Random Search simply chooses each sample time randomly, while Uniform Search and Voronoi Search keep track of the previously sampled locations, and cast the next sample at the midpoint of the largest unsearched region. The difference between Uniform and Voronoi searching lies in how the first sample is selected and how ties are broken between unsearched intervals of equal size. Uniform Search casts its first visibility sample at the midpoint of the exposure time interval and breaks ties by always selecting the earlier interval, resulting in a completely deterministic algorithm. Voronoi Search, on the other hand, casts the first sample at a randomly selected time, and breaks ties randomly as well.

Figure 3.19 gives code for implementing these three algorithms, with the "getTime" function determining the sampling pattern used. Since the samples can be thought of as partitioning the exposure time interval into smaller intervals, we assume the existence of a "Partition" class to keep track of the previous sample times. We require only that this class provide an "addSample" method to add a sample to the partition,

```
class Partition
{
public:
        addSample( float sampleTime );
        float[2] getLargestInterval( bool breakTiesRandomly );
        ...
};

float splitInterval()
{
        Partition curPartition;

        while( true ) {
                float curTime = getTime(curPartition)
                if( V(t_i) == SQUARE ) {
                        return curTime;
                } else {
                        curPartition.addSample(curTime);
                }
        }
}
```

Random Search:

```
float getTime( Partition p )
{
        return random01();
}
```

Uniform Search:

```
float getTime( Partition p )
{
        float interval[2] = p.getLargestInterval(false);
        return 0.5*(interval[0]+interval[1]);
}
```

Voronoi Search:

```
float getTime( Partition p )
{
        float interval[2] = p.getLargestInterval(true);
        if( This is the first sample ) {
                return interval[0] +
                        random01() * (interval[1]-interval[0]);
        } else {
                return 0.5*(interval[0]+interval[1]);
        }
}
```

Figure 3.19: Interval Splitting Algorithms.

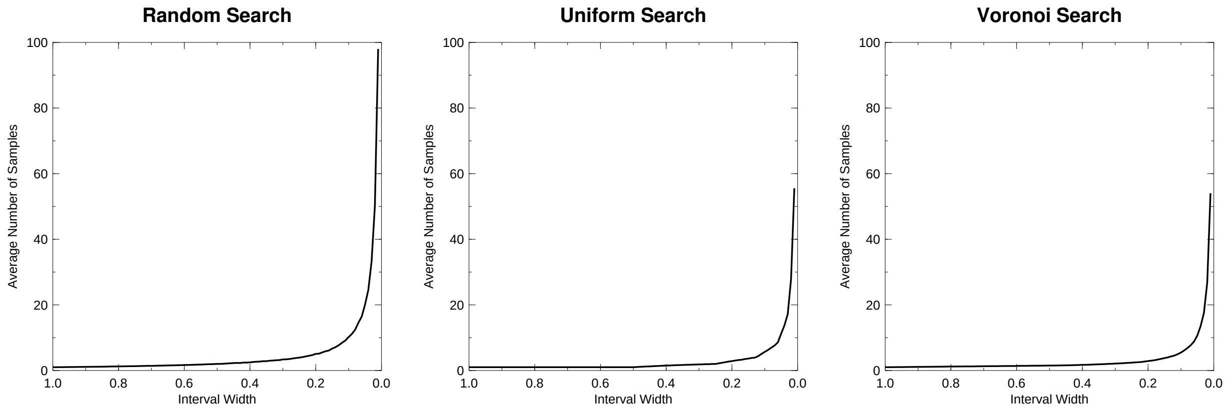


Figure 3.20: Convergence rates for Random Search, Uniform Search, and Voronoi Search. The average number of samples required to find a split time is graphed as a function of the width of the object's visible interval. The results are averaged over all possible locations of the visible interval.

and a "getLargestInterval" method to return the largest unsearched interval in the partition (with a boolean argument to indicate whether ties should be broken randomly). Figure 3.18 shows sample patterns generated using these algorithms. We will evaluate these algorithms based on two main criteria:

Convergence We would like to select the algorithm that finds a split time using as few visibility samples as possible.

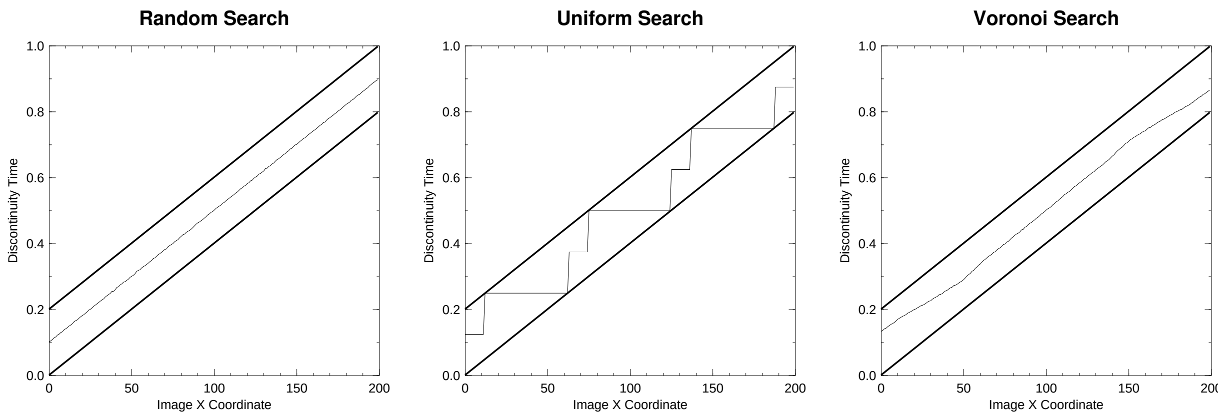
Midpoint Proximity For reasons that will become apparent later, we would like our split point to occur as close as possible to the midpoint of the visible interval. If the algorithm systematically produces a split point far from the midpoint of the interval, then artifacts will appear in the resulting images.

Convergence

Obviously, the number of samples required to locate a split time will depend on the width of the object's visible interval. In our example, the width of the visible interval is half of the width of the exposure time interval, but faster moving objects can produce far narrower intervals. Figure 3.20 examines the convergence behaviour of the three search strategies as the visible interval width is lowered. The figure demonstrates that Uniform Search and Voronoi Search converge at roughly the same rate, which is in turn faster than that of Random Search.

Midpoint Proximity

Since RSB has difficulty approximating the locations of discontinuities that are near the endpoints of the interval being searched, we would be wise to try to split the exposure time interval in a way that avoids this



5.00 samples/pixel **2.85 samples/pixel** **2.88 samples/pixel**

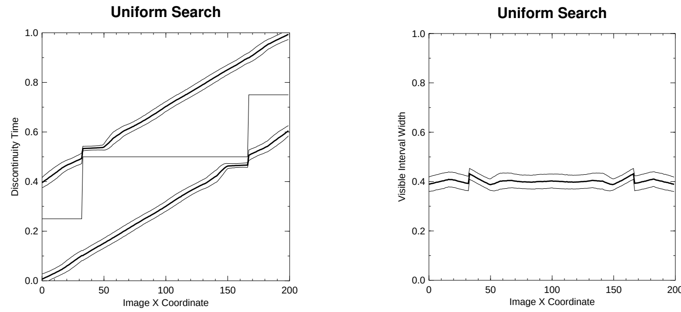
Figure 3.21: Expected split times for Random Search, Uniform Search, and Voronoi Search on a visible interval of width 0.2. The dark lines represent the exact locations of the discontinuities, and the lighter lines between them the expected split time, averaged over 5000 trials. The average number of samples required to find the split times are shown below each graph.

situation. The optimal solution would be to split the interval at the exact midpoint of the object's visible interval, as this would simultaneously maximize the distance of the left and right discontinuities from the endpoints of their respective subintervals. Thus, we would like our algorithm to produce split points that maintain a close proximity to the object's visible interval midpoint on average.

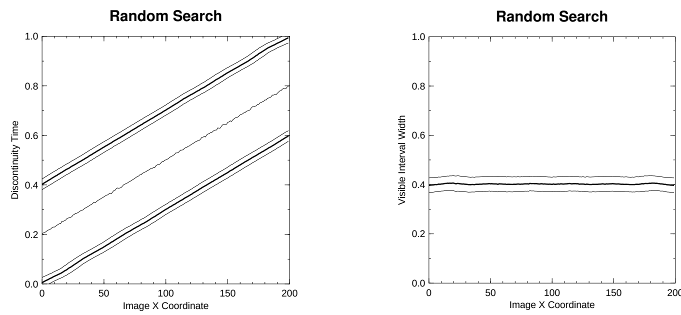
Figure 3.21 shows the split times produced by each algorithm on an example similar to our previous two-discontinuity example. The figure shows that Random Search produces the exact interval midpoint as a split point on average, but requires an average of 5 visibility samples per pixel to do so. Uniform Search, on the other hand, requires only 2.85 samples per pixel, but produces a split time that often deviates widely from the midpoint. Voronoi Search requires about the same number of samples as Uniform Search, but it produces split points that rarely deviate far from the visible interval midpoint. Thus, it would appear that Voronoi Search provides an optimal compromise between our requirements of fast convergence and midpoint proximity.

Conclusions

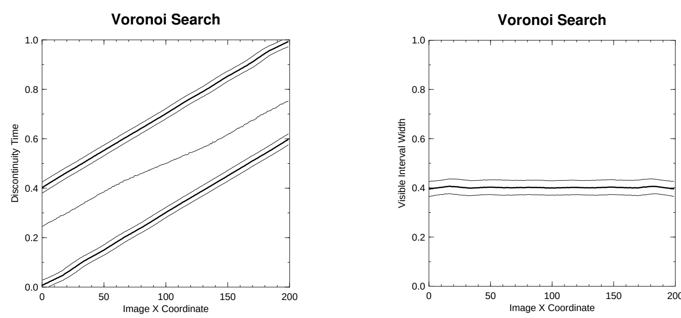
Figures 3.22 and 3.23 show the results of applying these algorithms to the two-discontinuity pixels in our test case. In Figure 3.22, the left column shows expectation/variance curves for the two discontinuity time estimates, as well as the average split time between them, as located by each algorithm. The right column shows the expectation and variance of the visibile interval width estimates produced by the algorithms. Figure 3.23 then shows sample images generated by the algorithms.



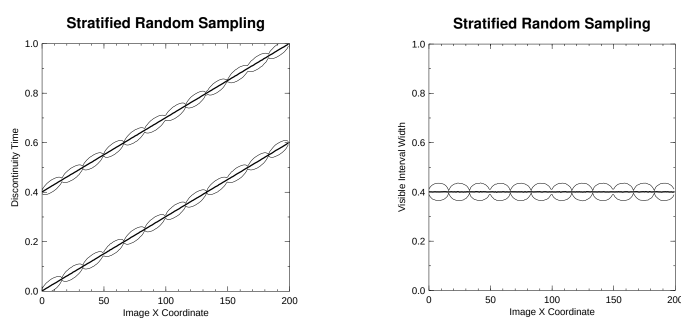
Uniform Search: 12.36 samples/pixel (1.50 US + 10.87 RSB)



Random Search: 13.20 samples/pixel (2.50 RS + 10.70 RSB)

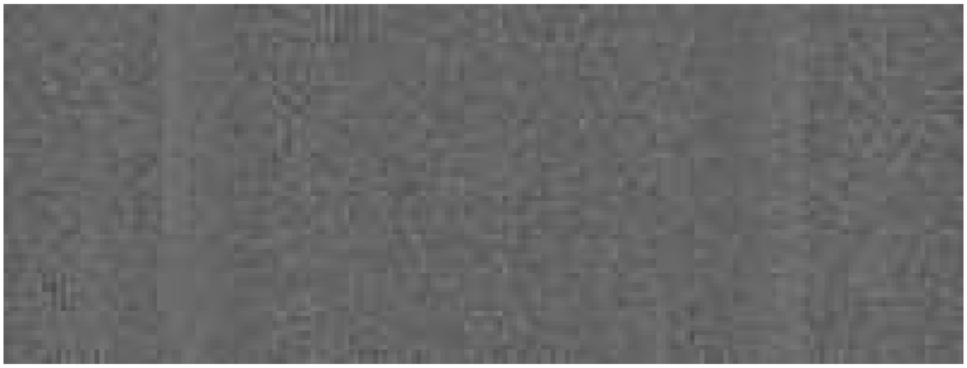


Voronoi Search: 12.49 samples/pixel (1.72 VS + 10.77 RSB)

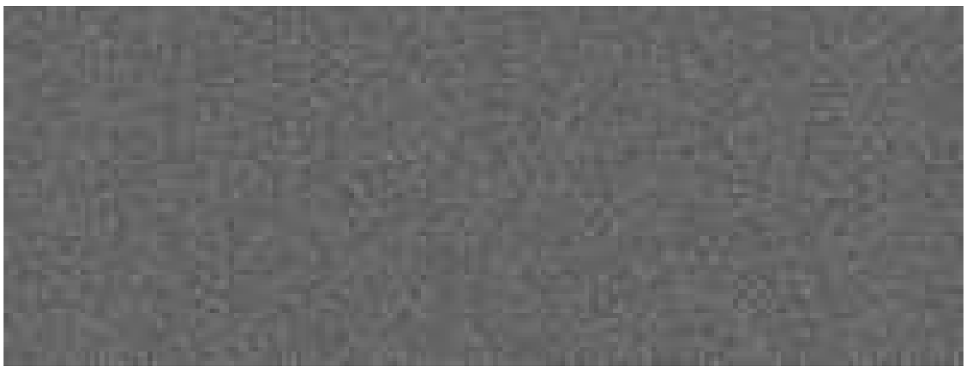


Stratified Random Sampling: 20 samples/pixel

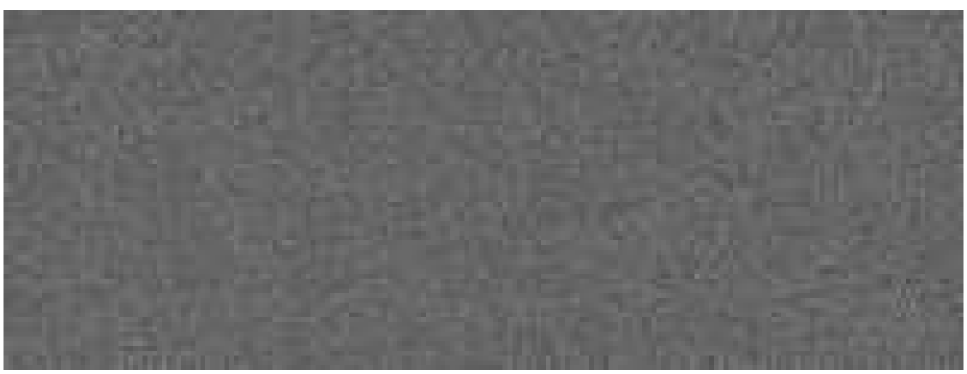
Figure 3.22: Two-discontinuity test case results. The left column shows the expectation/variance curves for the two discontinuity estimates, as well as the average split time separating them. The right column shows the expectation and variance curves for the resulting width estimate.



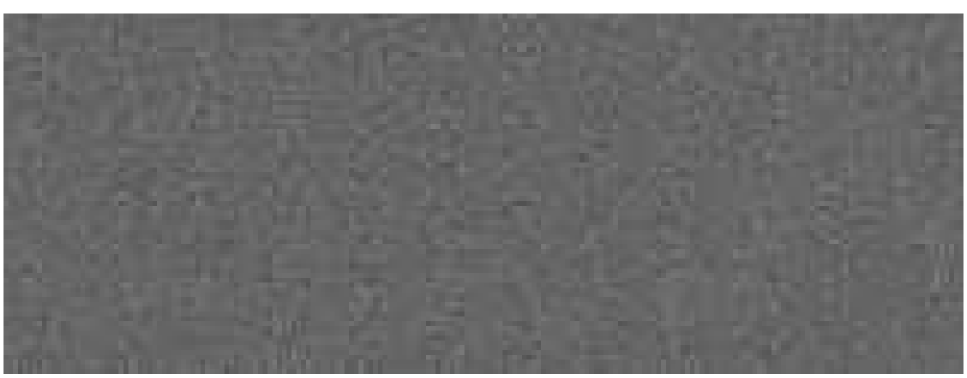
Uniform Search



Random Search



Voronoi Search



Stratified Random Sampling

Figure 3.23: Two-discontinuity images produced by the algorithms analyzed in Figure 3.22.

In these graphs, we see why midpoint proximity of the split time is so important. In the graph for Uniform Search, whenever the split time strays too far from the actual interval midpoint, it interferes with one of the discontinuity approximations, creating a "kink" in the expectation curve for that discontinuity. This causes two kinks in the expectation curve for the visible interval width estimate, which then translate directly into the artifacts observed in the resulting image. The other two splitting algorithms (Random Search and Voronoi Search) produce good split points, and therefore do not experience this problem, producing images free of artifacts. For comparison, the figure also shows a Stratified Random Sampling result that still exhibits noticeable banding artifacts, even using 20 samples/pixel (compared to 12.49 samples/pixel for VS/RSB). Thus, it appears that our adaptive strategy provides the best solution for pixels known to contain two discontinuities. In the following section, we describe the Two Discontinuity Finding (TDF) heuristic for handling pixels containing 0, 1, or 2 visibility discontinuities.

3.4.3 TDF - Putting it all together

In the previous sections, we have developed good algorithms for handling the cases where a pixel is known to have exactly one visibility discontinuity (RSB), or exactly two visibility discontinuities (VS+RSB). Ouellette and Fiume's Two Discontinuity Finding (TDF) heuristic incorporates both these algorithms to handle pixels in which the number of visibility discontinuities is not known in advance, but is assumed to be at most 2. TDF exploits pixel-to-pixel coherence in the visibility function to speed up the discontinuity finding process.

The Algorithm

Pseudocode for the TDF algorithm is given in Figure 3.24. First, the visibility at the endpoints of the exposure time interval $[0,1]$ are sampled. If $G(0) \neq G(1)$, the pixel must contain an odd number of discontinuities. Since we are assuming that there are at most two discontinuities, we conclude that the pixel must contain exactly one discontinuity in this case. RSB can therefore be used to estimate the discontinuity time.

If $G(0) = G(1)$, then there must be an even number of visibility discontinuities in the pixel, so we invoke Voronoi Search to decide whether the pixel contains 0 or 2 discontinuities. The search terminates either when a suitable split point t_s (with $G(t_s) \neq G(0)$) is found, or when the maximum unsearched interval size is less than a user-specified "Voronoi tolerance" ν. This guarantees that the search will terminate, even in pixels containing no visibility discontinuities (and hence no split point). If a split point is located, then the pixel must contain two discontinuities, so RSB is applied to locate the discontinuity times on opposite sides of the split point.

If the search terminates without finding a split point, we must conclude that the pixel contains no visibility

```
findDiscontinuities_TDF()
{
    Sample endpoint visibilities G(0), G(1)

    if( G(0) ≠ G(1) ) {
        Visibility function has a single discontinuity, so find it using RSB.

    } else {
        Visibility function has 0 or 2 discontinuities, so try to find a split point.
        Let P be number of discontinuities in previous pixel.

        if( P = 0 ) {
            No useful info from previous pixel, so use VS to search
                for a split point in the interval [t_0, t_1].

        } else if( P = 1 ) {
            Previous pixel has one visibility discontinuity (at time t_d, say),
                and one endpoint (t_e, say) has changed visibility.
            Find midpoint t_m between t_e and t_d, and sample visibility G(t_m).

            if( G(t_m) ≠ G(0) ) {
                Use t_m as a split point.
            } else {
                Use VS to search for a split point in [t_0, t_1].
            }

        } else if( P = 2 ) {
            Previous pixel has two visibility discontinuities, so sample visibility
                at midpoint t_m between the discontinuities.

            if( G(t_m) ≠ G(0) ) {
                Use t_m as a split point.
            } else {
                Use VS to search for a split point in [t_0, t_1].
            }
        }

        if( a split point t_s was found ) {
            Pixel has two discontinuities, one in [0, t_s],
            the other in [t_s, 1]. Use RSB to find each one.
        } else {
            Pixel has no discontinuities.
        }
    }
}
```

Figure 3.24: Pseudocode for the TDF algorithm.

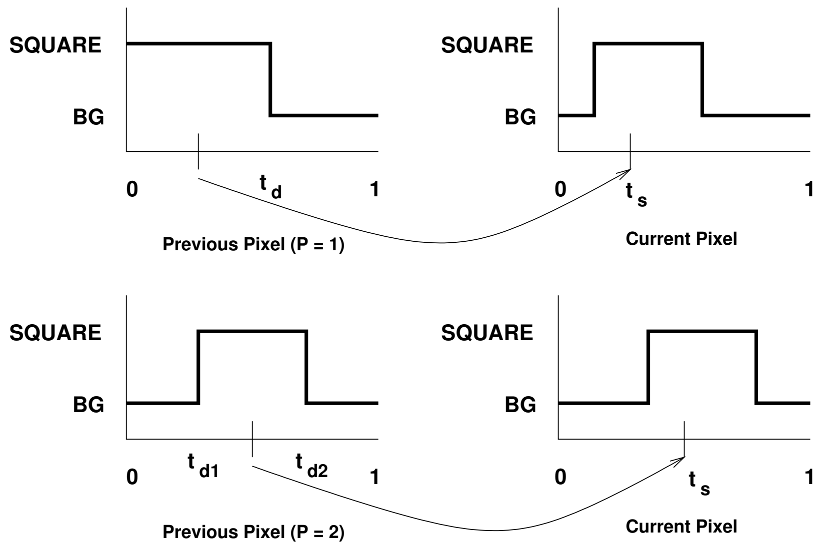


Figure 3.25: Illustration of how TDF computes the split point guess t_s based on the visibility function of the previous pixel. When the previous pixel had 1 discontinuity ($P = 1$), the guess is the midpoint between the discontinuity location t_d and the interval endpoint whose visiblility changed in moving from the previous pixel to the current one ($t = 0$ in this case). When the previous pixel had 2 discontinuities ($P = 2$), the guess is the midpoint between the discontinuities t_{d1} and t_{d2}.

discontinuities. This may not be true, however, as Voronoi Search may fail to locate a visible interval whose width is less than ν. Although we can minimize the impact of these missed intervals on the final pixel colour by lowering the tolerance ν, we do so at the cost of increased sample counts, especially in pixels containing no discontinuities. In practice, however, a relatively large tolerance (around 0.2) will suffice if we exploit pixel-to-pixel coherence in the visibility function by augmenting our search with visibility information from the previously processed pixel.

Pixel-to-pixel coherence in the global visibility function is readily apparent in our fast-moving square example from the previous section. Examining figure 3.16, we see that the object's visible interval shifts by only a small amount between adjacent pixels. As a result, if an object is visible at a split time t_s in one pixel, it will most likely still be visible at time t_s in an adjacent pixel. TDF uses this fact to guess an initial split point based on information from the previous pixel. Frequently, this first guess will succeed in locating a split point in the current pixel, thereby avoiding the need to invoke Voronoi Search altogether. If we let P represent the number of discontinuities found in the previous pixel, then there are three cases to consider:

Case $P = 0$. If the previous pixel contained no discontinuities, then there is no coherence information from which to guess a split point. TDF simply applies VS with a tolerance ν to try to locate a split point in the current pixel.

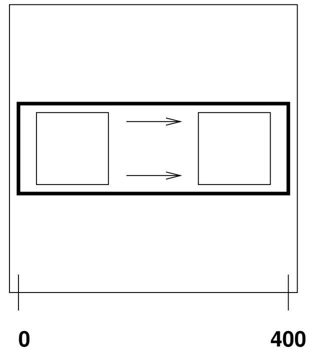


Figure 3.26: TDF test case. The highlighted region contains pixels having 0, 1, and 2 visibility discontinuities.

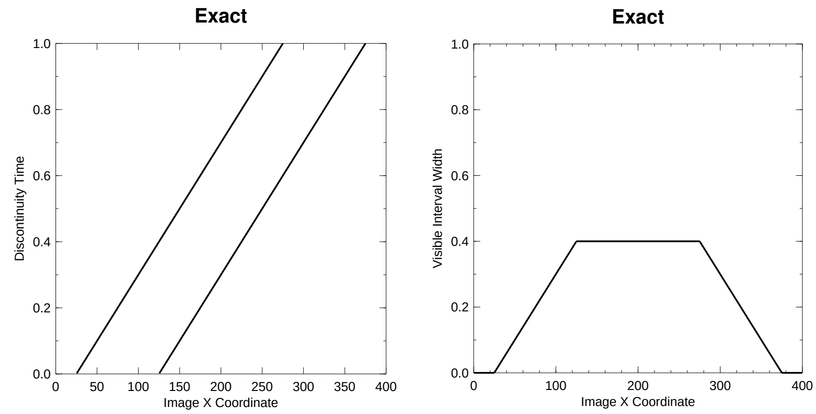
Case $P = 1$. If $G(0) \neq G(1)$ in the previous pixel, and $G(0) = G(1)$ in the current pixel, then the visibility at one of the endpoints of the exposure time interval ($t = 0$ or $t = 1$) must have changed. Let t_e be the endpoint whose visibility changed, and let t_d be the estimated discontinuity location from the previous pixel. TDF uses the midpoint between t_e and t_d as an initial split point guess t_s. If $G(t_s) \neq G(0)$, then the guess is appropriate, and the interval is split at t_s. Otherwise, VS is invoked to try to locate a split point. Figure 3.25 illustrates the relevant quantities for this case.

Case $P = 2$. If the previous interval had two discontinuities, at times t_{d1} and t_{d2}, then TDF uses the midpoint between the discontinuity times as an initial split point guess t_s. Again, if the guess succeeds ($G(t_s) \neq G(0)$), then the interval is split at this point, otherwise VS is invoked to try to find a split point. Figure 3.25 illustrates this case as well.

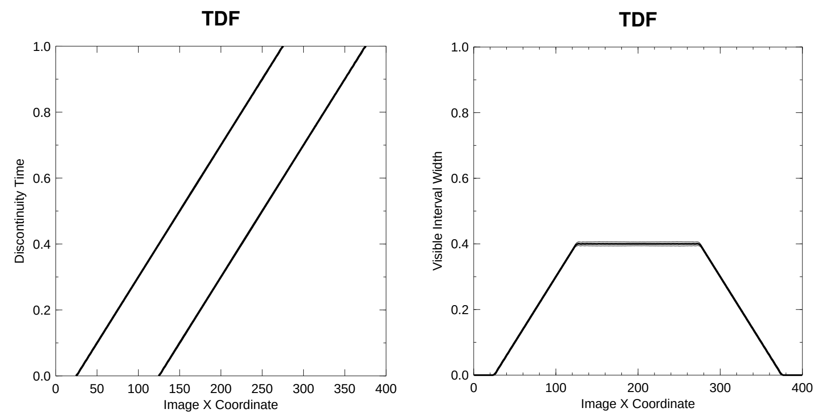
Results

To test the TDF algorithm, we apply it to a wider range of pixels in our moving square test case, as illustrated in Figure 3.26. Figure 3.27 shows the exact discontinuity times and interval widths, as well as the expectation and variance of the corresponding estimates produced by TDF (RSB tolerance $\epsilon = 0.02$ and Voronoi tolerance $\nu = 0.2$) and by 11-sample Stratified Random Sampling. Figure 3.28 shows the images produced by the algorithms. These results show that for around 11 samples/pixel, TDF produces a far superior image (almost indistinguishable from the exact result, in fact) than Stratified Random Sampling using an equivalent sample count.

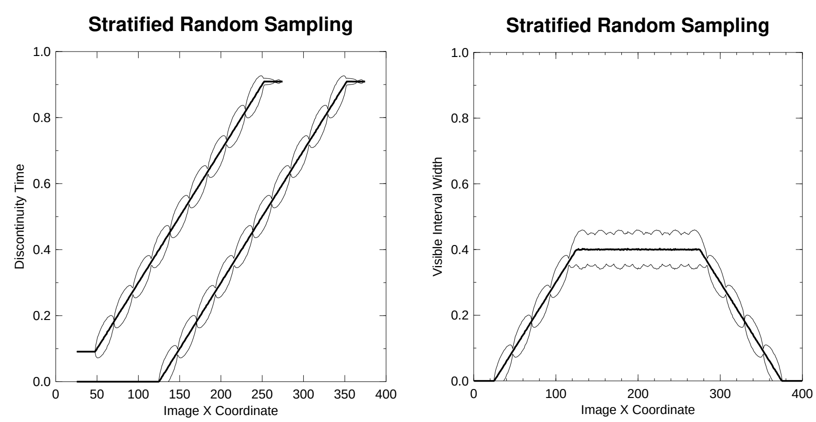
Figure 3.29 gives a breakdown of how many samples were used by the various stages of TDF. Obviously, each pixel required 2 samples to determine visibility at the exposure time interval endpoints. Discontinuity approximation using RSB accounted for another 7.62 samples per pixel, on average. Interestingly, just 1.1 samples per pixel were used in searching for a split time, including the initial guess suggested by the previous



Exact Discontinuity Times and Interval Widths



TDF ($\epsilon = 0.02$, $\nu = 0.2$): 10.72 samples/pixel

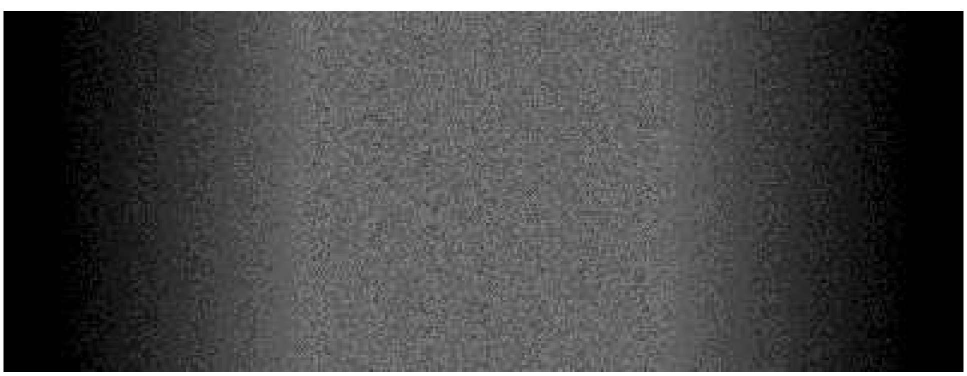


Stratified Random Sampling: 11 samples/pixel

Figure 3.27: Results of TDF and SRS, compared with the exact result for our test case. Using an RSB tolerance of 0.02 and a VS tolerance of 0.2, the TDF algorithm used an average of 10.72 visibility samples per pixel. The RS result was obtained using a comparable value of 11 samples per pixel. The TDF result is clearly superior.

Exact

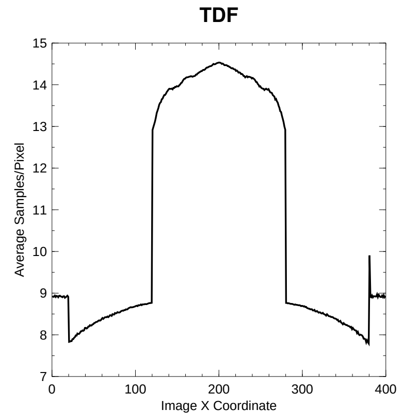
TDF



Stratified Random Sampling

Figure 3.28: Two-discontinuity images produced by the algorithms analyzed in Figure 3.27.

Purpose	Number of Samples
Endpoint Sampling	2
Split Time Search	1.10
Random Seed Bisection	7.62
Total	**10.72 samples/pixel**



Sample Count Breakdown **Per-Pixel Average Sample Counts**

Figure 3.29: Sample count analysis for TDF. The table shows the number of samples per pixel used for VS, RSB, and endpoint sampling, averaged over all pixels in the image. The figure on the right shows how the average sample count varies as a function of image position.

pixel. The reason for this is that this initial split point guess was almost always correct, due to the high degree of pixel-to-pixel coherence in the visibility function. Only rarely was it necessary to invoke Voronoi Search to locate a split point.

The graph in Figure 3.29 shows the average number of visibility samples as a function of position in the image. The first 25 pixels contain no discontinuities, so VS had to be invoked, resulting in an average of about 9 samples/pixel being cast. The next hundred pixels have a single discontinuity, so only endpoint sampling and RSB were necessary, resulting in 8-9 samples per pixel being cast. Pixels 125-275 all encounter 2 visibility discontinuities, so the average sample count roughly doubles, since RSB must be invoked on each discontinuity. Finally, the last 125 pixels are symmetric to the first 125, except for a spike at pixel 375, where we transition between 1 and 0 visibility discontinuities. At this point, there is no split point to be found, but the previous pixel information produces an initial split point guess which TDF tries before invoking VS. As a result, one extra sample per pixel is cast.

Thus, we have seen that in the presence of 0, 1, or 2 visibility discontinuities, the TDF algorithm produces a far more pleasing result better than Stratified Random Sampling using an equivalent number of samples.

3.5 Conclusions

In this chapter, we have examined the discontinuous temporal visibility functions that result from scenes involving moving objects. We saw how motion–blurred images can be generated by approximating the location of visibility discontinuities using point–sampling. We observed the excessive noise and banding artifacts inherent in the traditional Stratified Random Sampling approach, then showed that these artifacts can be alleviated using strategies based on Ouellette and Fiume's Random Seed Bisection algorithm. Using the related TDF heuristic, we can produce smoothly blurred images for scenes containing up to 2 visibility discontinuities per pixel using fewer visibility samples than are required by Stratified Random Sampling. In the next chapter, we will develop the *Multiple Discontinuity Finding* (MDF) algorithms for producing similar results in the presence of an arbitrary number of visibility discontinuities.

Chapter 4

Temporal Visibility for an Arbitrary Number of Discontinuities

The TDF algorithm described in the previous section was originally designed for computing illumination from a one-dimensional "linear" light source. It is important in this setting to compute which segments of the light source are visible from a particular point P in space. A visibility function $\{V(l) : l \epsilon [0, 1]\}$ is defined along the light source, with $V(l) = 1$ if point l on the light source is visible from point P, and $V(l) = 0$ if point l is occluded. TDF is used to locate discontinuities in $V(l)$, and from these, the visible segments on the light source can be approximated. By integrating the radiance emanating from these segments of the light source, the total incident illumination at point P can be computed.

This problem bears a great resemblance to the motion blur problem as we have formulated it in this chapter. In both problems, we wish to locate discontinuities in a one-dimensional integer-valued visibility function. The difference between the two problems lies in the exact nature of this visibility function. For the linear lighting case, the visibility function $V(l)$ can take one of only two values, 0 or 1, and is assumed to have at most two discontinuities over its domain, an assumption that appears to hold in the majority of cases.

Unfortunately, the global visibility function $G(t)$ used in the motion blur problem frequently violates these assumptions. It can assume as many values as there are objects in the scene, and it is common for multiple objects to pass through a pixel during the exposure time interval. A set of N objects crossing a pixel can give rise to as many as $2N$ visibility discontinuities if all objects are all convex and moving linearly, and an unbounded number of discontinuities if they are not (consider a spinning helicopter blade - by speeding up its rotation or by lengthening the exposure time interval, we can make the blade pass through a given

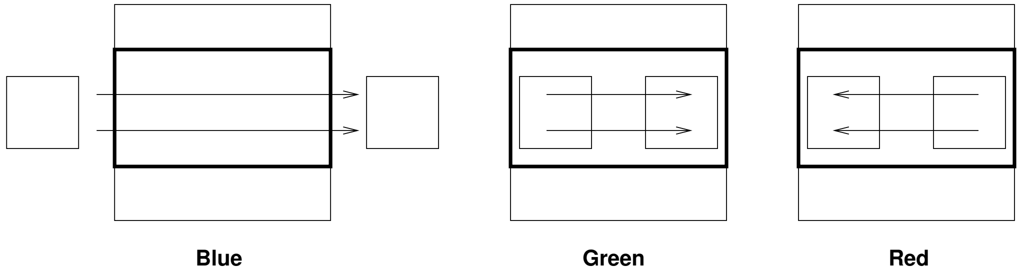


Figure 4.1: MDF test scene. The back-to-front depth ordering of the objects is red, green, blue.

pixel centre as many times as we like!). Thus, in the motion blur domain, there is a definite need for an algorithm that will handle an arbitrary number of visibility discontinuities produced by an arbitrary number of objects. This section describes a set of *Multiple Discontinuity Finding* (MDF) algorithms, which satisfy this requirement.

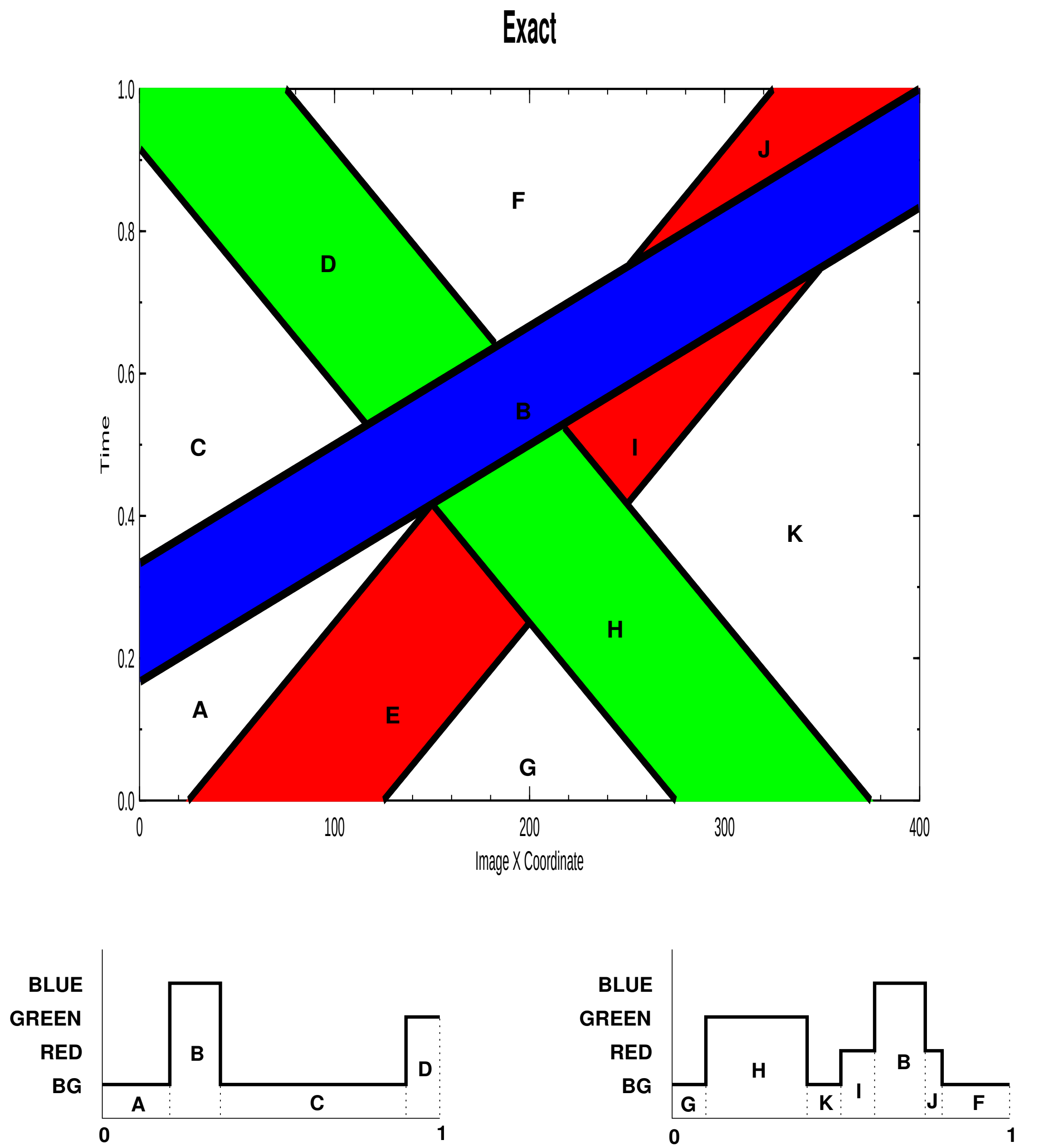
In Section 4.1, we will describe an example of a scene containing multiple discontinuities per pixel, which we will use to test our algorithms. In Section 4.2, we describe MDF1, the basic algorithm for handling an arbitrary number of discontinuities. In Section 4.3, we describe how to augment MDF1 with visibility information from the previous pixel, resulting in the MDF2 algorithm. In Section 4.4, we further exploit pixel-to-pixel coherence by adding *backtracking* to MDF2 to produce the MDF3 algorithm. Finally, Section 4.5 presents a complicated test case on which to evaluate the final MDF3 algorithm.

4.1 Multiple Discontinuity Example

Before we describe the MDF algorithms, we will first present an example of a scene whose visibility function frequently contains more than two discontinuities. In the scene (shown in Figure 4.1), three squares of equal size are moving horizontally in image space. We assume that the size and depth ordering (blue, green, red from front to back) remain constant over the course of the exposure time interval.

To visualize the visibility function for this scene, we use a *visibility graph*, shown in Figure 4.2. The visibility graph is simply a colour-coded spacetime diagram in which the colour of a point (x, t) denotes the object visible through the centre of pixel x at time t. To construct the exact visibility function for a particular pixel centre x_c, we simply take a vertical slice through the graph at $x = x_c$, as is shown in the figure for pixels $x = 0$ and $x = 260$.

The visibility discontinuities (shown as dark lines in the graph) partition the graph into a number of spacetime regions of constant visibility. By labelling these regions A through K, we can classify each pixel's visibility function according to the regions it intersects (and hence the visible intervals that it contains). We



Global Visibility Function for $x = 0$ **Global Visibility Function for $x = 260$**

Figure 4.2: Visibility graph for our test scene. The visibility function for each pixel can be extracted by taking a vertical slice through the graph (as shown for pixels $x = 0$ and $x = 260$). The letters A to K label spacetime regions in which a particular object is visible, and can thus be used to classify the structure of the visibility function for a particular pixel.

can use these classifications to discuss structural changes in the visibility function as we move through the image. For example, pixel $x = 0$ is an (A, B, C, D) pixel. As we move to the right through the image, the A and C intervals become narrower, and eventually disappear. Meanwhile, the D interval moves to earlier times, the B interval moves to later times, and new G and F intervals appear at the start and end of the exposure time interval. By the time we get to pixel 260 (a (G, H, K, I, B, J, F) pixel), the visibility function bears little resemblance to what we started off with, but we can concisely describe the sequence of structural changes that led to it.

Figure 4.3 shows an exact motion-blurred image of the scene as well as a "discontinuity image" in which pixels are colour-coded according to the number of visibility discontinuities they contain. Although not as detailed as the visibility graph, the discontinuity image is useful as a high-level image-space overview of the structure of the visibility function. Both the visibility graph and the discontinuity image will prove useful tools in evaluating the algorithms developed in this chapter.

4.2 MDF1 - The Basic Multiple Discontinuity Finding Algorithm

The TDF algorithm discussed in the last section is in essence a two-stage strategy. The algorithm first uses Voronoi Search to determine how many visibility discontinuities (and hence how many visible intervals) are contained in the pixel, then it uses Random Seed Bisection to approximate the location of the endpoints of these intervals. When the visibility function contains at most 2 discontinuities, TDF makes the following three guarantees about the resulting solution:

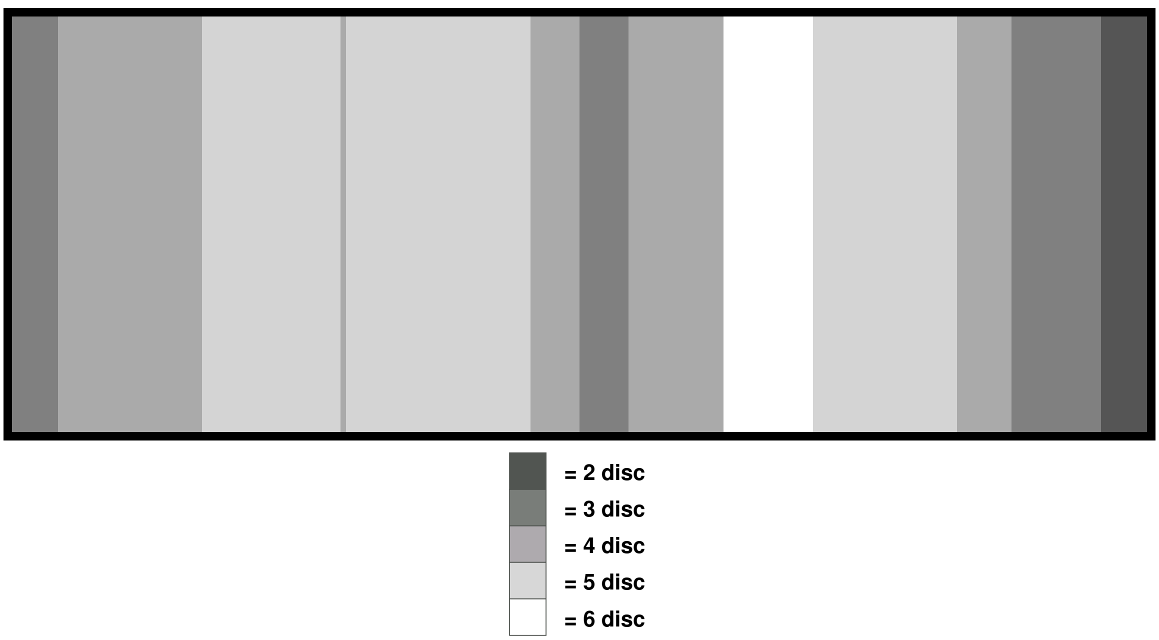
Guarantee 1: No visible interval whose width is greater than a specified tolerance ν is missed.

Guarantee 2: No visible interval endpoint approximation deviates by more than a specified tolerance ϵ from the actual endpoint location.

Guarantee 3: All visible interval endpoint approximations vary smoothly over the image.

The first two guarantees result from the termination conditions of the Voronoi Search and Random Seed Bisection algorithms, respectively. The third guarantee is ensured by the combination of the continuous-expectation, constant-variance nature of RSB and the tendency for VS to locate suitable split points. We will now use the ideas of Voronoi Search and Random Seed Bisection to formulate our first Multiple Discontinuity Finding algorithm (MDF1), which will provide similar guarantees, but in the presence of an arbitrarily complex visibility function.

Exact Motion-Blurred Image



Exact Discontinuity Image

Figure 4.3: Exact motion-blurred image and discontinuity image for the sample scene. The discontinuity image shows the number of visibility discontinuities at each pixel, and therefore indicates where changes in the structure of the visibility function occur in the image.

```
void Partition::finishPartition( float ε, float ν )
{
      while( there exists an unfinished segment [t_i, t_{i+1}] ) {
            Compute visibility at segment midpoint t_mid
            Add this visibility sample to the partition
      }
}
```

Figure 4.4: Pseudocode for the finishPartition operation.

As with Voronoi Search in the previous section, MDF1 uses a Partition data structure to keep track of visibility samples that are cast. Central to the MDF1 algorithm is the idea that a set of temporal visibility sample times $\{t_0 = 0, t_1, \ldots, t_N = 1.0\}$ partition the exposure time interval $[0, 1]$ into a set of disjoint segments $\{S_1 = [t_0, t_1], \ldots, S_N = [t_{N-1}, t_N]\}$. Starting from an initial partition consisting of a single segment $[0, 1]$ covering the entire exposure time interval, we wish to add visibility samples to the partition, thereby subdividing the segments, until each segment $[t_i, t_{i+1}]$ in the partition satisfies **one of** two criteria:

RSB Criterion: $G(t_i) \neq G(t_{i+1})$ and $\mid t_{i+1} - t_i \mid < \epsilon$, **or**

VS Criterion: $G(t_i) = G(t_{i+1})$ and $\mid t_{i+1} - t_i \mid < \nu$.

We will refer to a segment satisfying one of these criteria as a *finished* segment. Once all segments in the partition are finished, the visibility approximation constructed from the samples in the partition will clearly meet our first two guarantees, since any interval of width greater than ϵ and containing a single discontinuity will be subdivided, as will any interval of width greater than ν that contains an entire visible interval (i.e. 2 discontinuities). Assuming the existence of a Partition data structure, we can easily formulate a simple **finishPartition** operation (listed in Figure 4.4) that repeatedly adds visibility samples to the partition at the midpoints of unfinished segments, terminating when all segments in the partition are finished.

Although it will finish all the segments in the partition, and thus meet the first two guarantees, this strategy will certainly not approximate interval endpoints in a continuous fashion, and thus will violate the third guarantee. The reason for this is that the finishPartition operation is a deterministic search strategy, in that new samples are always added to the partition at the midpoints of unfinished segments. Our experience with RSB and VS has clearly shown that purely deterministic search strategies will produce artifacts (banding or otherwise) in the resulting images. These artifacts can be eliminated, however, by incorporating a bit of randomness at the start of the search. Thus, our MDF1 algorithm operates as follows: starting from an initial partition consisting of a single segment $[0, 1]$, we first insert one visibility sample taken at a randomly

```
void Partition::findDiscontinuities_MDF1( float ε, float ν )
{
    Compute visibility at a randomly chosen time t_r ε [0, 1]
    Add this visibility sample to the partition
    finishPartition( ε, ν )
}
```

Figure 4.5: Pseudocode for the MDF1 algorithm.

chosen time, then we invoke the finishPartition operation. Pseudocode for this algorithm is given in Figure 4.5.

Figure 4.6 shows sampling patterns produced by MDF1 on two visibility functions, using a Voronoi tolerance $\nu = 0.2$ and an RSB tolerance $\epsilon = 0.05$. In the figure, we differentiate between RSB samples, which are cast in segments whose endpoint visibilities differ, and VS samples, cast in segments whose endpoint visibilities are the same. In both cases, the initial sample R is placed at the randomly chosen time $t = 0.4$.

In the simple example on the left, the RSB samples (1-3) zoom in on the location of the discontinuity in an RSB fashion, starting from the initial random time R. Thus, MDF will produce a good approximation of this discontinuity location. Meanwhile the VS samples (A-D) are busy searching the rest of the exposure time interval in the manner of Voronoi Search, again starting from the random time R. Since no other visible intervals are found, the MDF algorithm terminates, having cast a total of 10 visibility samples.

This result is identical to the sampling pattern that would be produced by TDF on this example, except for the extra VS samples. Because of the restrictions it places on allowed visibility functions, TDF would assume that the known discontinuity is the only one in the pixel, and thus leave the region between R and 1.0 unsearched, confident that no other visible interval can exist there. In the general case, however, we can no longer make this assumption. We must guarantee that no visible interval wider than $\nu = 0.2$ is missed, so we must search any segments that are wider than this, as they may contain such a visible interval.

To demonstrate this, consider the second visibility function presented in the figure. Here, we have added an extra visible interval of width 0.2. Since the pixel now contains 3 discontinuities, TDF could not handle this case properly. At best, it would produce good estimates of some of the discontinuities, while ignoring the others, and at worst it could become confused and produce one or more bad discontinuity approximations. MDF1, however, can deal with this case effectively. The segment $[0, R]$ is searched exactly as in the previous case, resulting in a good RSB approximation of the first discontinuity location, and an assurance that no other large visible intervals exist in this region. Meanwhile, the segment $[R, 1.0]$ receives a single VS sample,

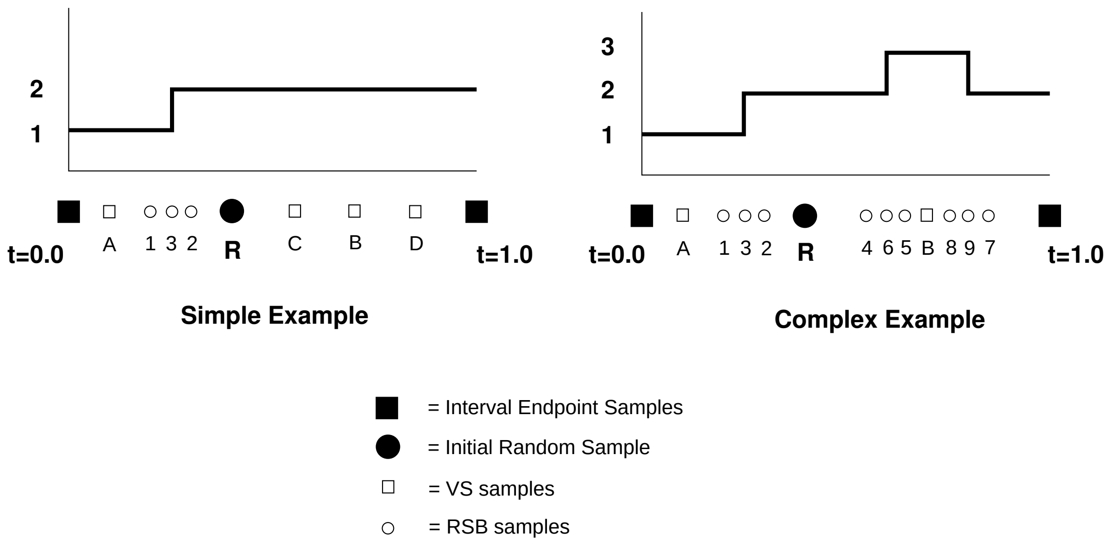


Figure 4.6: Examples of MDF1 sampling patterns. RSB samples are labelled with numbers, and VS samples with letters. In the simple case, MDF1 reduces to RSB, while in the complex case, MDF1 reduces to Random Seed Bisection for the first discontinuity, and to a combination of Voronoi Search followed by Random Seed Bisection for the second and third discontinuities.

B, that identifies the presence of the visible interval for object 2. A number of RSB samples (4-6 and 7-9) are cast to zoom in on the discontinuities in the segments $[R, B]$ and $[B, 1.0]$, respectively. At this point, all segments satisfy the RSB or VS criteria, hence MDF terminates, having located all the visible intervals.

In order to satisfy our third guarantee, we must ensure that all discontinuities are approximated in a continuous fashion. In our example, we have seen that MDF1 is equivalent to RSB in searching for the first discontinuity, so this discontinuity is well-approximated. We can show that this is also the case for the next two discontinuities. We saw in a previous section that the use of Voronoi Search to find a split point between two discontinuities followed by Random Seed Bisection on either side of the split point will produce continuous estimates of both discontinuities. We will now demonstrate that MDF approximates the second and third discontinuities in an equivalent manner.

In the example, the VS point B acts as a split point between the discontinuities. The sequence of samples that led to B being selected is equivalent to that produced by Voronoi Search (starting from the initial random sample position R), so we know that B is a suitable split point. We now must verify that MDF conducts an RSB-like search on either side of this split point. This is less obvious, since it would appear that the sample sequences used (4,5,6 and 7,8,9) are both deterministic. This is not the case, however, due to the fact that samples 4 and 7 are both indirectly derived from the position of the initial random sample R, and hence are themselves random. Thus, as in RSB, the search for each discontinuity proceeds deterministically from a random initial position, so we are guaranteed that the approximation will be continuous throughout the image.

Thus, MDF1 approximates each discontinuity location in accordance with the principle of deterministic searching from a random starting sample, which our earlier investigations have identified as being essential to the production of a visually satisfying result. It is true that all the discontinuity approximations for a particular pixel are correlated (due to the fact that all random sample positions are derived from a single initial random sample position R), but this does not appear to adversely affect the results. It would be far more worrisome if the solutions for adjacent pixels were somehow correlated, as such correlations would most likely produce noticeable artifacts in the resulting images. Although not a formal proof, we feel that the arguments presented thus far are sufficient to conclude that the solution produced by MDF1 meets all three of our guarantees.

4.2.1 MDF1 Results

Having explained the MDF1 algorithm, we now test it on our sample scene. Figure 4.7 displays the results produced using MDF1 with tolerances of $\nu = 0.2$ and $\epsilon = 0.03$, requiring an average of about 20 visibility samples per pixel. The visibility graph shows the expectation and variance of the discontinuity estimates, superimposed over the exact colour-coded visibility solution for reference. For each discontinuity, the variance of our estimate is so small as to be almost unnoticeable on the graph, resulting in the computed image being free of banding artifacts or excessive noise. Figure 4.8 presents the same information for 20-sample Stratified Random Sampling. The SRS visibility graph shows the expected variance banding of the discontinuity estimates, resulting in excessive noise and banding artifacts in the computed image. For the same number of samples, the MDF1 image is clearly superior. Figure 4.9 shows the sample count breakdown for the MDF1 case.

The MDF1 image is not perfect, however, as it contains some noise that we would not expect given our low RSB tolerance. The noise is noticeable in the regions marked 1 and 2 in the error image in Figure 4.7. In region 1, green speckles are present in the predominantly dark area, while orange speckles are visible in the otherwise green area of region 2. This noise is not the result of MDF1 poorly approximating the endpoints of a known visible interval, rather it is the result of intervals being missed entirely. We should expect to see many such errors, since we are only guaranteed to locate intervals wider than $\nu = 0.2$, and the visibility graph shows that many pixels contain one or more intervals narrower than this tolerance. The discontinuity image in Figure 4.7 shows the number of discontinuities actually located by MDF1 in each pixel, and the corresponding “error image” highlights the pixels in which one or more intervals are missed. As expected, the error image shows that intervals are frequently missed in many regions of the image.

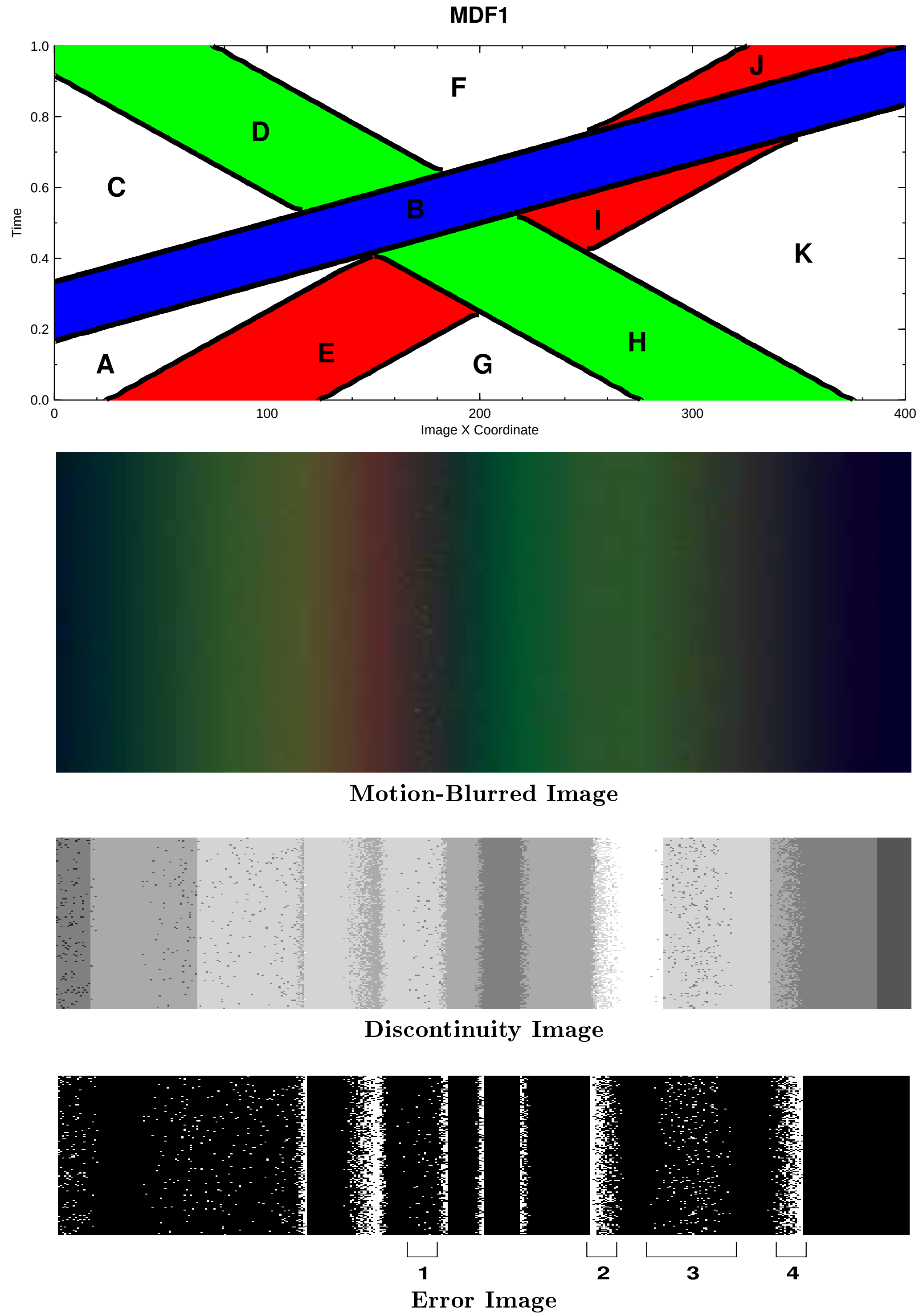


Figure 4.7: MDF1 results. The visibility graph shows the expectation and variance of the discontinuity approximations. Along with the final motion-blurred image, the figure shows a discontinuity image indicating the number of discontinuities found at each pixel, as well as an "error image" highlighting the pixels in which this discontinuity count differs from the actual number of discontinuities. Marked regions 1 and 2 contain errors that are noticeable in the final image.

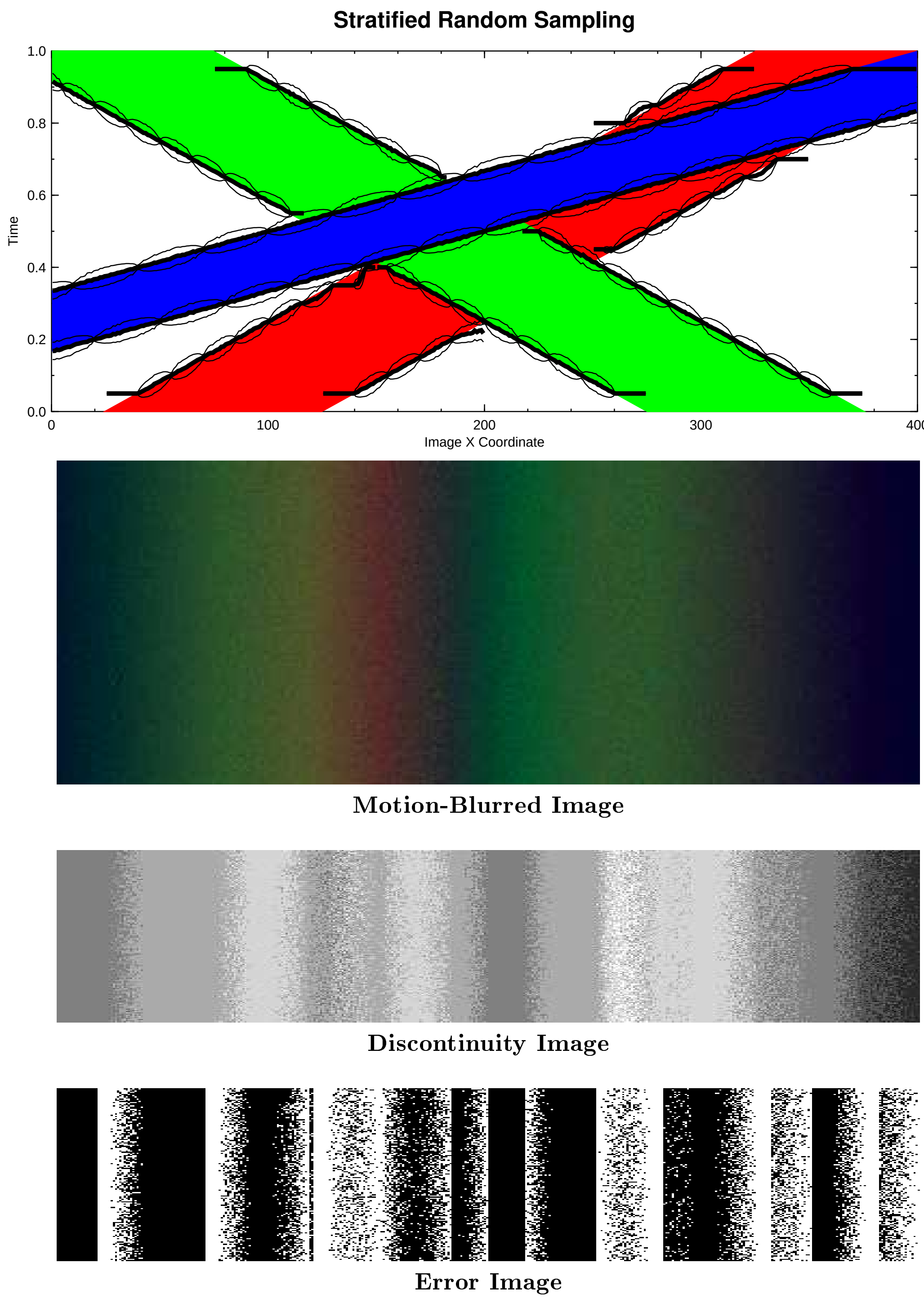
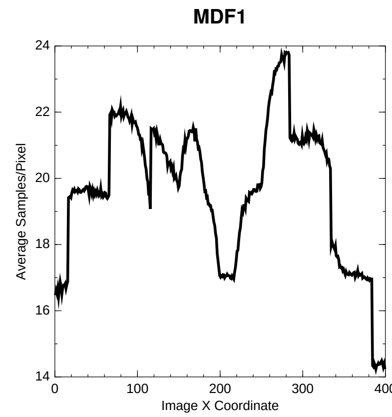


Figure 4.8: Stratified Random Sampling Results (20 samples/pixel).

Purpose	Number of Samples
Endpoint Sampling	2
Initial Random Sample	1
VS Samples	1.52
RSB Samples	15.13
Total	**19.66 samples/pixel**



Sample Count Breakdown **Per-Pixel Average Sample Counts**

Figure 4.9: Sample count analysis for MDF1. The table shows the number of samples per pixel used for VS, RSB, and endpoint sampling, averaged over all pixels in the image. The figure on the right shows how the average sample count varies as a function of image position.

Many of these errors result from the fact that the visible intervals being missed are extremely thin. For example, the errors in Region 3 in the error image are the result of I intervals shrinking to zero width near $x = 350$ in the image. These errors do not concern us, as the intervals being overlooked in this case are so narrow that they have little impact on the final pixel colour, and thus do not contribute any artifacts to the final image. Similar effects occur at numerous other locations in the image.

More troubling, however, are the errors in Regions 1 and 2 in the error image, which are the result of missing B intervals, which although narrower than the Voronoi tolerance, are still wide enough to make a significant contribution to the final pixel colour. In Region 1, missing the B interval causes its area to be attributed to the green object instead of the blue, resulting in a pixel that is too green. Similarly, in Region 2, the B interval is sandwiched by I and J intervals, so if the B interval is missed, its contribution is attributed to the red object, resulting in a pixel that is too red. These are the only errors that contribute noticeable artifacts to the final image.

In this case, these artifacts can be eliminated if we take advantage of the image space coherence of the visibility function, just as TDF did. In the next section, we describe the MDF2 algorithm, which is simply MDF1 augmented with coherence information from the visibility solution for the previous pixel.

4.3 MDF2 - Incorporating Information from Previous Pixels

The TDF algorithm made effective use of pixel-to-pixel coherence in the visibility function, using the structure of the previous pixel's visibility function to guess at the visibility structure of the current pixel. The exact

```
void Partition::findDiscontinuities_MDF2( float ε, float ν )
{
    Let N be the number of visible intervals in the previous pixel,
    Let M_i denote the midpoint of the i^th visible interval in the previous pixel.

    foreach i in [2,N-1] {
        Sample visibility at time M_i and add result to the partition.
    }

    if( left endpoint (t = 0) changed visibility ) {
        Sample visibility at time M_1 and add result to the partition.
    }

    if( right endpoint (t = 1) changed visibility ) {
        Sample visibility at time M_N and add result to the partition.
    }

    foreach segment s in the partition {
        if ( segment s is not finished ) {
            Sample visibility at a random time in segment s and add to partition
        }
    }

    finishPartition( ε, ν )
}
```

Figure 4.10: Pseudocode for the MDF2 algorithm.

visibility solution presented in Figure 4.2 shows that our multiple-discontinuity example exhibits a similar degree of visibility coherence, which is not exploited by the MDF1 algorithm. In this section, we augment MDF1 with previous-pixel information, resulting in the MDF2 algorithm.

Figure 4.10 presents pseudocode for our new MDF2 algorithm. Instead of casting a single initial random sample into the partition, we now cast samples at the midpoints of the visible intervals computed in the previous pixel. The midpoints of the first and last intervals are used only if the visibility changes at the corresponding endpoint of the exposure time interval ($t = 0$ or $t = 1$) between pixels. In the case of pixels having at most two visibility discontinuities, this strategy produces the same initial guess as TDF. Figure 4.11 shows an example of two hypothetical visibility functions for adjacent pixels, and the previous pixel interval midpoints that are used to begin sampling the current pixel. The last midpoint M_5 is used because the visibility at $t = 1$ changes between pixels, while the first midpoint M_1 is not used because the visibility at $t = 0$ does not change.

After sampling visibility at the endpoints and inserting the previous pixel midpoints, we have an "initial

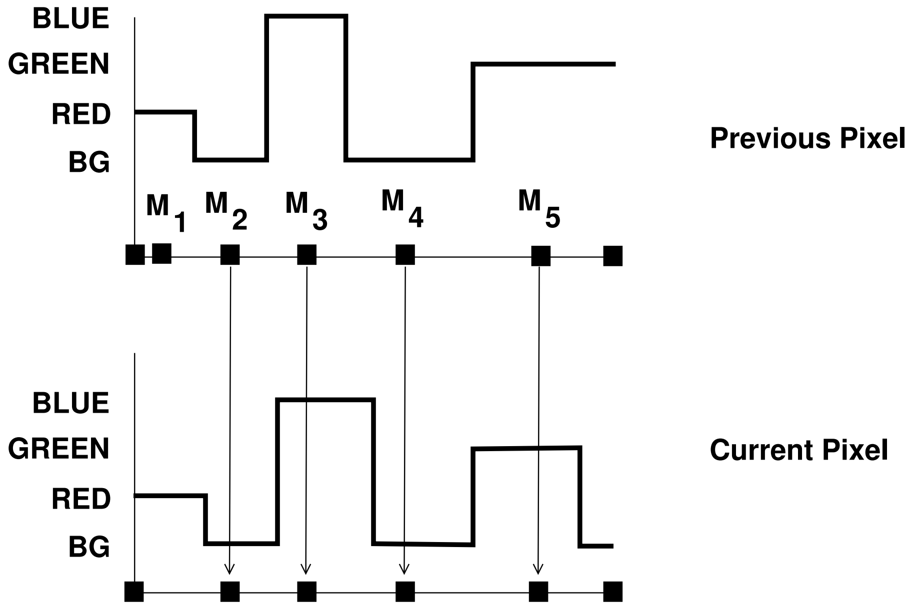


Figure 4.11: Previous pixel information used by MDF2. The midpoints M_2, M_3, M_4, M_5 of the intervals from the previous pixel are inserted into the partition for the current pixel. The first interval midpoint M_1 is not inserted because the visibility of the left endpoint of the exposure time interval ($t = 0.0$) does not change between the pixels.

partition" that most likely has identified all of the visible intervals in the current pixel (as is the case with the example in Figure 4.11), but we are not ready to invoke **fillPartition** just yet. As was explained in the previous section, we must introduce some randomness into the partition before the deterministic fillPartition is called, to ensure that the resulting approximation satisfies our third guarantee. The visible interval midpoints from the previous pixel are not random quantities, so they do not qualify. To provide the necessary randomness, we loop through the unfinished segments of the initial partition and randomly place a visibility sample in each. Once this is done, we invoke fillPartition to compute the final solution. In essence, this algorithm is just invoking MDF1 on each of the initial segments, thus we are confident that the solution will satisfy all the guarantees made by that algorithm.

4.3.1 MDF2 Results

Figure 4.12 shows the results of applying MDF2 to our sample scene. The error image shows that the problems with missing B intervals in regions 1 and 3 have been eliminated, and the resulting image is free of the speckling artifacts observed earlier. The vast majority of remaining errors (in regions 2 and 4, for example) are due to extremely small intervals being missed, and therefore are not noticeable in the final image. The reason why MDF2 misses these tiny intervals, even with the help of coherence information from

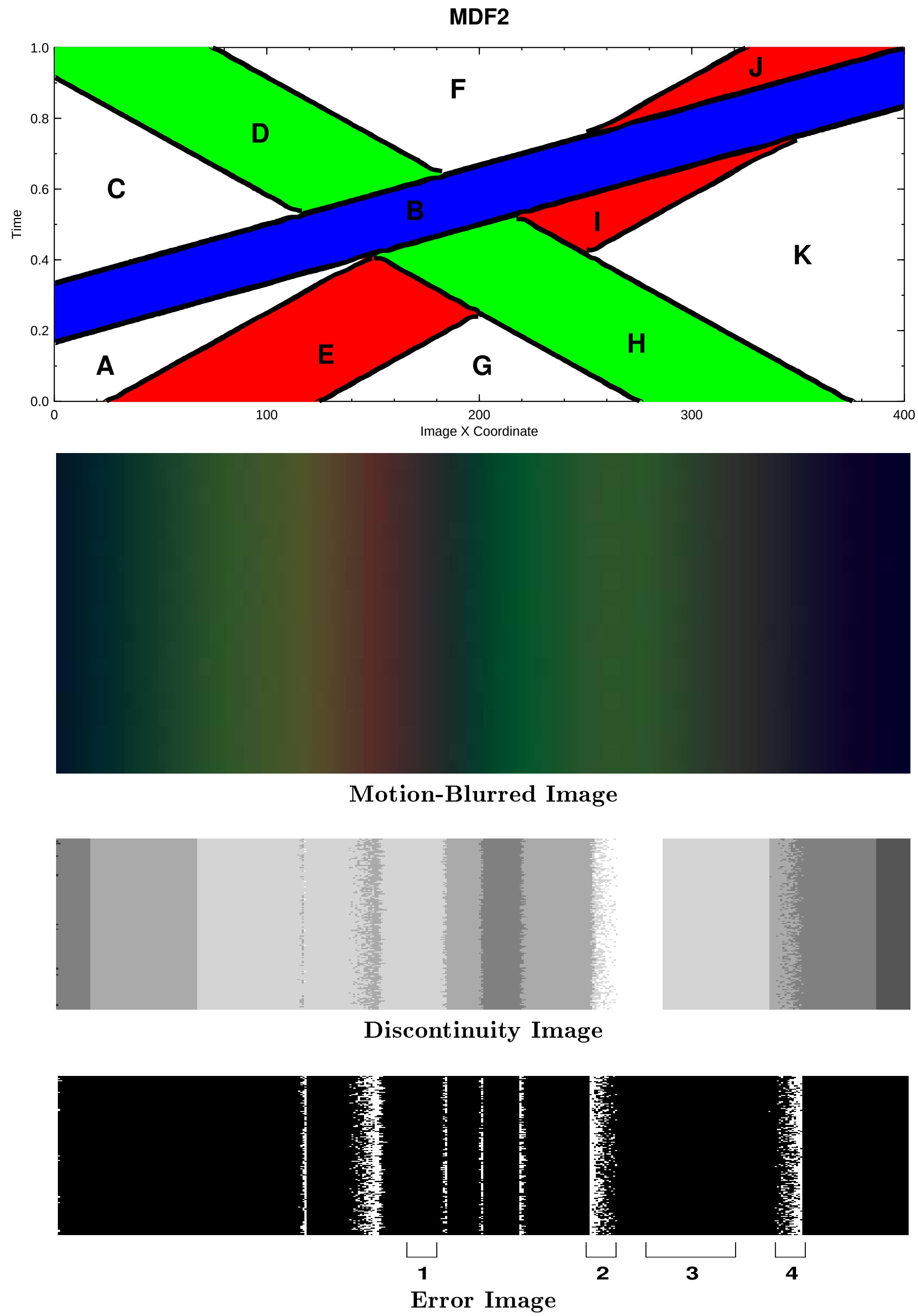
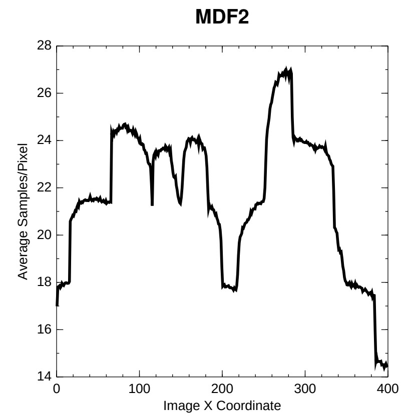


Figure 4.12: MDF2 Results.

Purpose	Number of Samples
Endpoint Sampling	2
Coherence Samples	3.30
VS Samples	0.55
RSB Samples	15.82
Total	**21.67 samples/pixel**



Sample Count Breakdown **Per-Pixel Average Sample Counts**

Figure 4.13: Sample count analysis for MDF2. The table shows the number of samples per pixel used for VS, RSB, coherence, and endpoint sampling, averaged over all pixels in the image. The figure on the right shows how the average sample count varies as a function of image position.

the previous pixel, is that the coherence information available is merely an approximation of the previous pixel's visibility function. When the width of a visible interval approaches the error in this approximation (the RSB tolerance ϵ), the midpoint guess from the previous pixel may be inaccurate, causing the interval to be missed in the current pixel.

Unfortunately, these tiny intervals cause slight problems even when they are found, as demonstrated by the slight wiggles in the expectation curves in regions 2 and 4 in the visibility graph. These wiggles are analogous to the problems observed with the Uniform Search algorithm in Section 3.4.2, where we saw that if a split point is chosen too close to a visible interval endpoint, RSB-style searching will have difficulty approximating that endpoint. Being derived from Voronoi Search, MDF2 tends to choose split points near the centres of visible intervals, so this problem is avoided most of the time. When a visible interval is extremely narrow (width $< \epsilon$), though, there may exist no suitable split point, so we are doomed to a poor approximation of its endpoints. We feel that this problem can be safely ignored, however, as the intervals being poorly approximated are so narrow as to have little impact on the final pixel colour. If necessary, the effects of the problem can be minimized by lowering the RSB tolerance, at the cost of a few extra samples per pixel. At the tolerance used for the test case ($\epsilon = 0.03$), the impact of these errors is not noticeable in the final image.

Figure 4.13 shows the sample count distribution for this example. MDF3 requires about 2 extra samples per pixel over MDF1. An average of 3.3 "coherence samples" (previous pixel interval midpoints and initial random samples) are cast per pixel. There is a decrease in the number of VS samples required (about 1 fewer sample per pixel), due to the fact that in most cases, the coherence information identifies all the visible

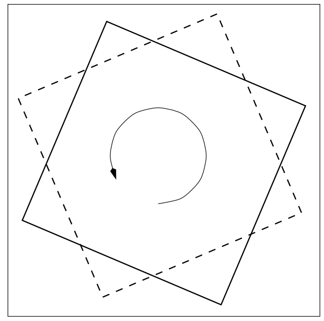


Figure 4.14: Backtracking test scene.

intervals in the pixel, so not much Voronoi searching is required.

Thus, we have observed that tangible benefits can be obtained from exploiting image space coherence in the visibility function at the cost of only a few samples per pixel. In the next section, we develop the MDF3 algorithm, which exploits coherence information from pixels other than the previously processed pixel.

4.4 MDF3 - Exploiting Coherence through Backtracking

The MDF2 algorithm described in the previous section exploits the assumption that for each visible interval in a pixel, there probably exist visible intervals of similar widths and positions in adjacent pixels. Using this idea of pixel-to-pixel coherence, MDF2 is frequently able to locate visible intervals narrower than the Voronoi tolerance ν, such as the B intervals in our previous example. Once a small interval is found in a pixel, it will most likely also be found for subsequent pixels, since coherence is propagated in the same order that pixels are processed. If we were to also propagate this information backwards along a scanline, we could also use information from the current pixel to identify intervals that were missed in previous pixels. In this section we propose the MDF3 algorithm, which is simply MDF2 plus coherence propagation to previously processed pixels, which we term **backtracking**. After describing a test case that demonstrates the need for backtracking, we will formulate the MDF3 algorithm and apply it to the test case.

4.4.1 Backtracking Example

The need for backtracking can be illustrated using the test scene shown in Figure 4.14. The scene consists of a white square that rotates through 45 degrees counterclockwise against a black background. Figure 4.15 shows an exact motion-blurred image of this scene, as well as results produced by MDF1 and MDF2 applied to the test case, using tolerances of $\nu = 0.3$ and $\epsilon = 0.02$.

The limitations of MDF1 and MDF2 are exposed in the corners of the image, where the spinning corners of the object cause 2 visibility disconitinuities as they sweep through pixels. Since the scene is vertically symmetric, we focus only on the lower-left and lower-right corners. The zoomed images of these corners show that MDF1 deviates from the exact result in both corners, while MDF2 only has difficulty with the lower left corner. The discontinuity images shown in Figure 4.16 show that these errors are the result of the algorithms failing to identify the spinning object's visible intervals in these regions of the image.

To see why the visible intervals are missed, we turn once again to a visibility graph. Figure 4.17 shows the visibility graph for scanline 30 in the image. We observe in the graph two spacetime regions A and B in which the object is visible. These regions consists of pixels through which the lower left and lower right corners of the object sweep, respectively. Each region contains a highlighted subregion in which the visible intervals are narrower than the Voronoi tolerance $\nu = 0.3$, and hence are not guaranteed to be found by MDF1 or MDF2.

In these narrow areas, MDF1 will sometimes locate and sometimes miss the object's visible interval, resulting in the speckling effect observed in the final image. Since each pixel is processed independently, the speckles are randomly distributed throughout the troublesome regions. The MDF2 image, on the other hand, exhibits a different set of artifacts. The discontinuity image appears identical to the exact solution, except that a series of horizontal chunks have been taken out of the lower left corner. Once again, the visibility graph explains these errors. The correct result is produced in the lower right corner because coherence from the large B intervals between $x = 100$ and $x = 200$ lead the algorithm to the small B intervals between $x = 200$ and $x = 225$. For the lower left corner (region A), however, no such information is available to the algorithm. It must rely on luck to find the narrow intervals between $x = 40$ and $x = 60$. Once a single narrow interval is located, however, coherence will lead the algorithm to the intervals in all subsequent pixels, hence the "chunk-like" nature of the errors in this region.

Since previous-pixel coherence produces an ideal solution for the B region, we know by symmetry that a backtracking scheme should solve the problems with finding the A intervals. The following section outlines the MDF3 algorithm, which is MDF2 augmented with backtracking.

4.4.2 MDF3 Algorithm

In order to implement backtracking, we must slightly alter the way in which we process pixels. With MDF1 and MDF2, once a pixel's visibility function approximation has been constructed, it is immediately used to compute the pixel colour, which is written into the image. With MDF3, we may wish to refine a pixel's visibility function approximation based on results from as-yet unprocessed pixels. Thus, we must store each

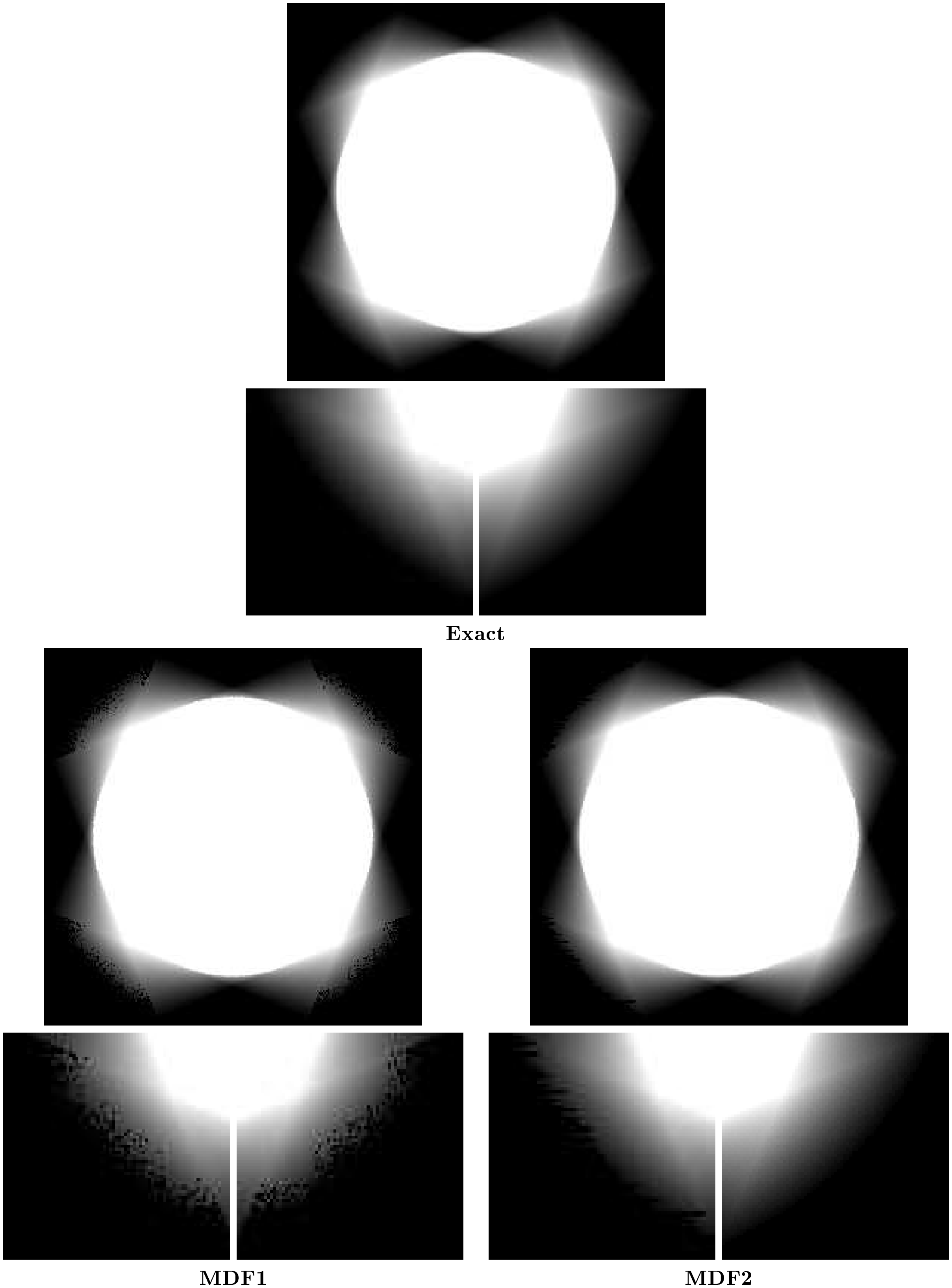


Figure 4.15: Exact motion-blurred image for the test scene, along with images produced by MDF1 and MDF2. The zoomed images of the bottom left and right corners demonstrate the nature of the errors made by these algorithms.

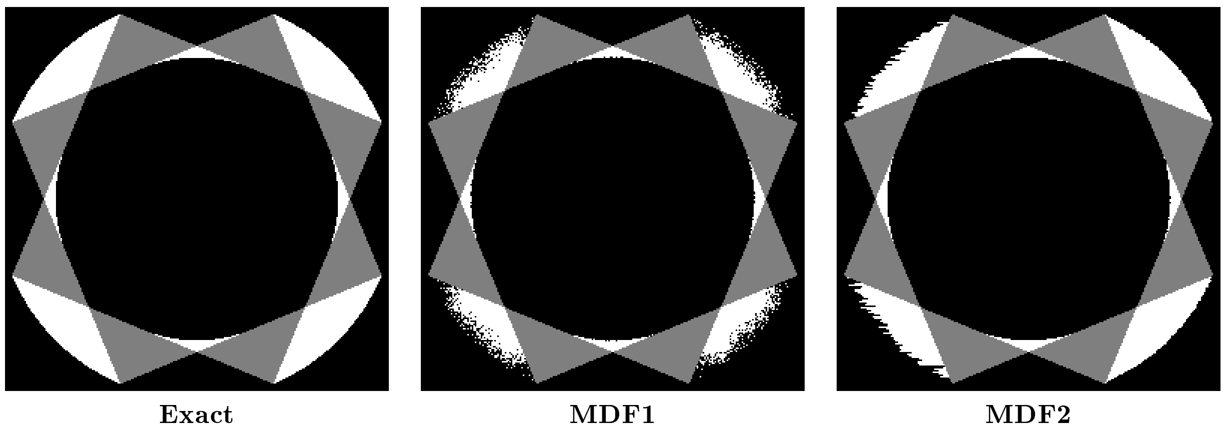


Figure 4.16: Discontinuity images produced by MDF1 and MDF2, along with an exact solution for reference. Black represents 0-discontinuity pixels, grey represents 1 discontinuity, and white represents 2 discontinuities.

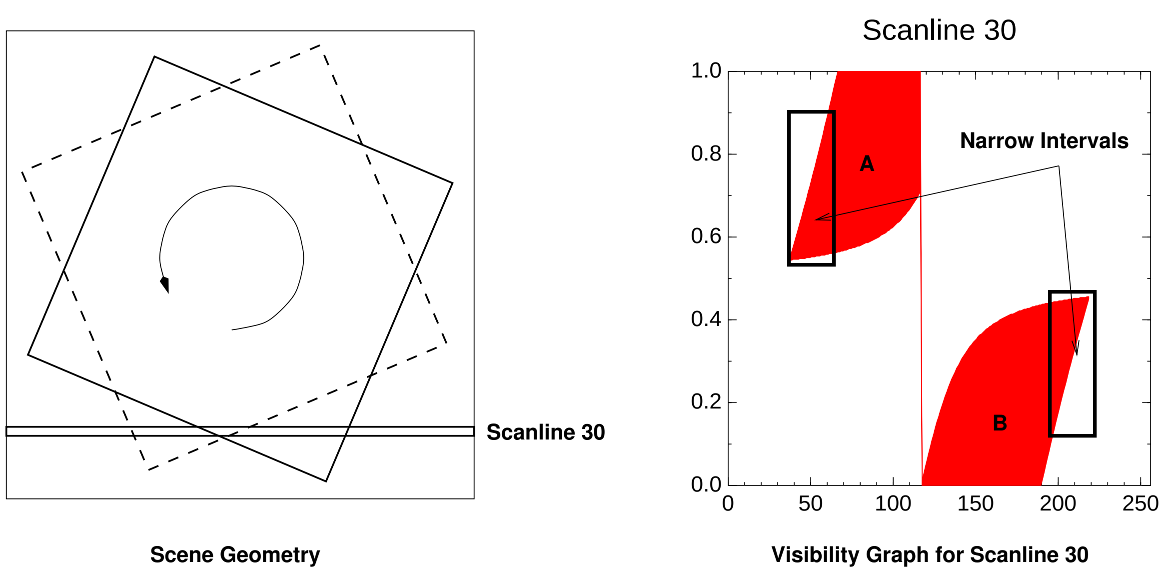


Figure 4.17: Visibility graph for scanline 30 of our test scene. The two highlighted regions contain intervals narrower than the Voronoi tolerance $\nu = 0.3$.

pixel's visibility approximation, committing it to the image only when we are convinced that it is accurate.

To accomplish this, we introduce the first-in-first-out (FIFO) *visibility cache*, which stores the visibility partition structures for previously processed pixels. As each pixel is processed, we propagate any relevant coherence information back through the partitions stored in the cache. The cache has finite size, so pixels will eventually be forced out of it, at which point their colours are computed and committed to the image. We have found that a cache size of about 25 pixels is reasonable in most situations.

Figure 4.18 gives a pseudocode description of the algorithm. To process a pixel, we first invoke MDF2 to generate a visibility partition for the pixel, from which a set of visible intervals are extracted. Each interval I_{small} whose width is less than the Voronoi tolerance ν is then "propagated" through the cache. Propagation involves searching through previously processed pixels for intervals similar to I_{small}, which we believe should exist due to coherence in the visibility function. First, we take the midpoint t_{mid} of the small interval and determine the objects visible at that time in the current and previous pixels (according to the partitions P_{cur} and P_{prev}). If the visible objects $P_{cur}(t_{mid})$ and $P_{prev}(t_{mid})$ in these partitions are different, then there is a good chance that an interval similar to I_{small} has been missed in the previous pixel (remember, the partition P_{prev} is merely an approximation to the true visibility function of the previous pixel). We sample visibility in the previous pixel at time t_{mid} to compute the object O_{prev} that is actually visible at that time. If this visibility sample indicates that an interval has been missed in the previous pixel (i.e. $O_{prev} \neq P_{prev}(t_{mid})$), then we add the new sample to the partition, and call fillPartition on P_{prev} to approximate the endpoints of the newly found interval. This new interval is then propagated back to the next pixel in the cache, and the process continues. If, on the other hand, the new visibility sample offers no new information about the previous pixel (i.e. $O_{prev} = P_{prev}(t_{mid})$), then we terminate our traversal of the cache.

Figure 4.19 shows a hypothetical sequence of 4 consecutive pixels, each containing a single narrow interval, and illustrates how MDF3 processes them. In steps 1 and 2 in the figure, the first two pixels are processed by MDF3, and the resulting partitions added to the cache. We assume that the narrow interval in each pixel is missed during this initial processing. In step 3, we process the third pixel, and by chance locate the narrow interval. This interval is then propagated to the previous pixel in the cache, which leads to the narrow interval in pixel 2 being discovered. This interval is then propagated back to pixel 1 in the cache, where another narrow interval is located. Having reached the end of the cache, we terminate our cache traversal. In step 6, the fourth pixel is processed, and the narrow interval is located. In propagating this interval, MDF3 realizes that the previous pixel already knows about it, so it is likely that all the other pixels in the cache know about it as well (due to the backtracking from pixel 3 when it was first processed). MDF3 therefore terminates its cache traversal. By ensuring that only new information is propagated through the

```
void findDiscontinuities_MDF3( float ε, float ν )
{
    Create new partition P_cur for the current pixel.
    P_cur.findDiscontinuities_MDF2( ε, ν )

    foreach small interval I_small in the current pixel
        Compute midpoint t_mid for the interval
        Let P_cur(t_mid) be the object visible at this time in the current pixel

        foreach partition P_prev in the cache
            Compute visible object P_prev(t_mid) in previous partition
            if P_prev(t_mid) = P_cur(t_mid) {
                // The small interval exists in the previous pixel,
                // so stop traversing the cache
                break;
            } else {
                // The interval does not exist in the previous pixel,
                // so sample the visibility in the previous pixel
                // at the expected interval midpoint.
                Cast a visibility sample to compute the actual visible object O_prev
                    at time t_mid in the previous pixel
                if ( O_prev ≠ P_prev(t_mid ) {
                    // A new interval has been discovered in the previous pixel
                    Add the new visibility sample to P_prev
                    P_prev.fillPartition()
                    Set t_mid to be the midpoint of the newly found interval
                    continue; // keep propagating through the cache
                } else {
                    // The new sample provided no new information, so halt propagation
                    break;
                }
            }
        }
    }

    Insert partition P_cur into the cache
    For pixel P_exit leaving the cache, compute its color and commit it to the image.

}
```

Figure 4.18: Pseudocode for the MDF3 algorithm.

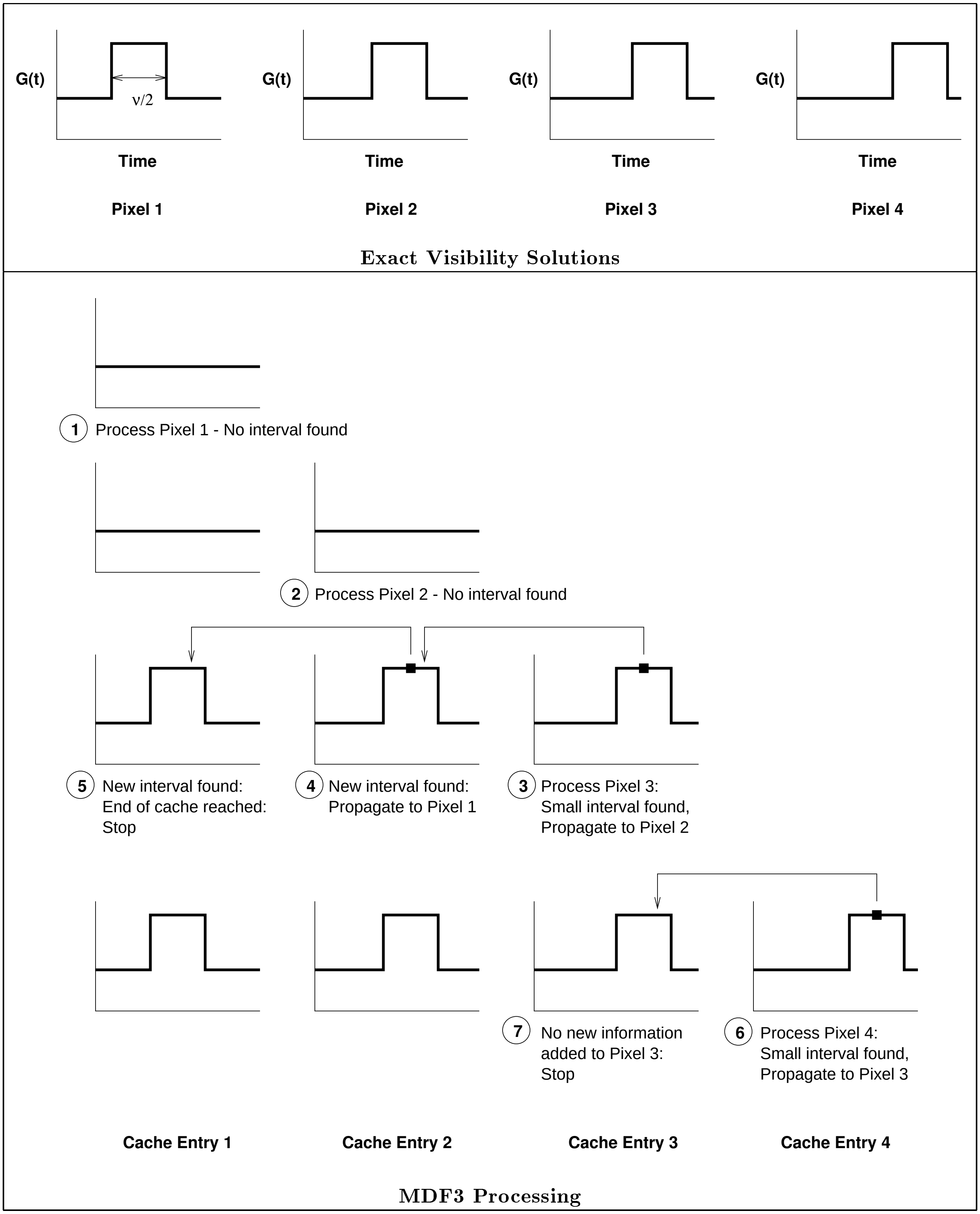
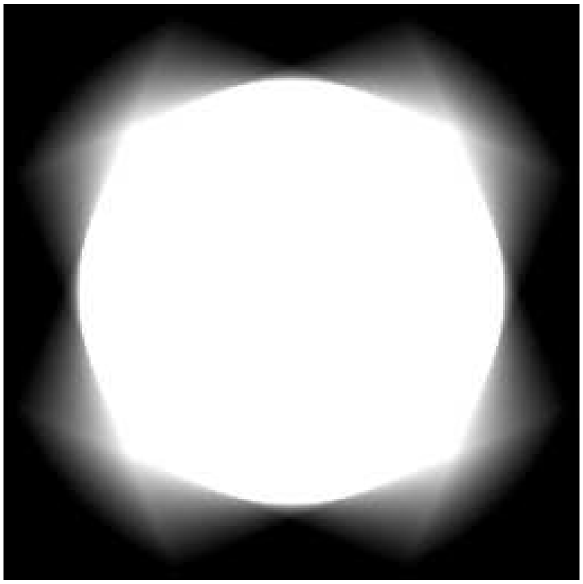
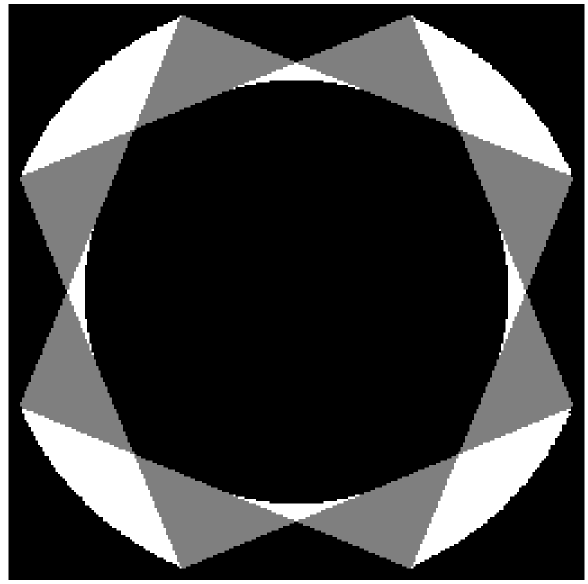


Figure 4.19: MDF3 processing example, illustrating the steps involved in processing four pixels whose exact visibility solutions are shown at the top of the figure. As each pixel is processed, it is added to the cache, and its small intervals are propagated through the other cache entries.



Motion-Blurred Image



Discontinuity Image

Purpose	Number of Samples
Endpoint Sampling	2
Previous-Pixel Coherence Samples	0.81
VS Samples	1.83
RSB Samples	2.64
Backtracking Samples	0.01
Total	**7.38 samples/pixel**

Sample Count Breakdown

Figure 4.20: Results of the MDF3 algorithm. All artifacts have been eliminated, and the result is indistinguishable from the exact result. The sample count breakdown shows that the cost of incorporating backtracking is only 0.01 samples per pixel, on average.

cache, a lot of redundant traversal is avoided. We now examine the results of MDF3 on our test scene.

4.4.3 MDF3 Results

Figure 4.20 shows the motion-blurred and discontinuity images produced by MDF3 on our test case. The images show that the problems with the left corners have been eliminated, and the resulting image is almost indistinguishable from the exact result. The accompanying sample count breakdown shows that the cost of adding backtracking is a mere 0.01 extra samples per pixel, on average.

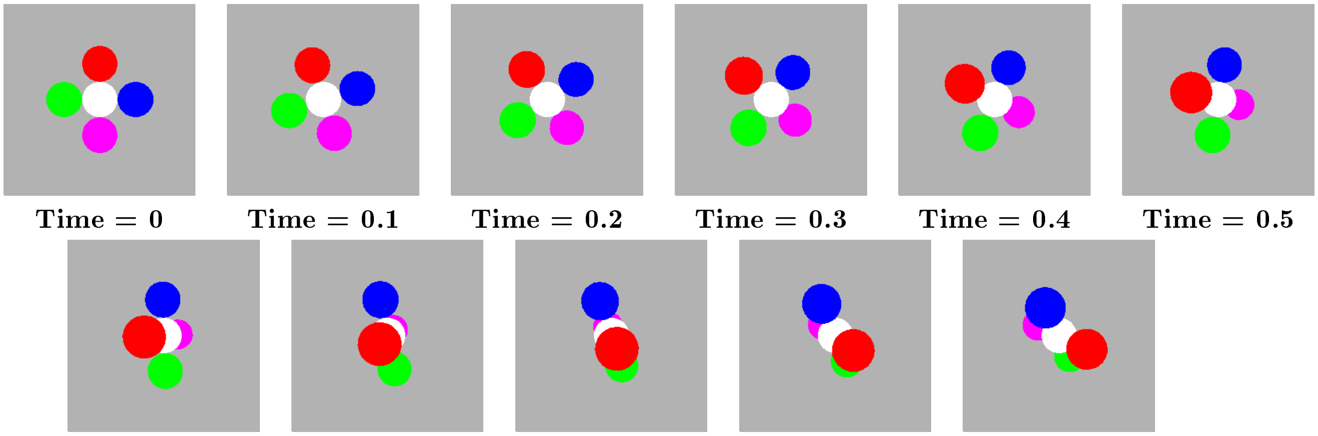


Figure 4.21: Another MDF3 test case.

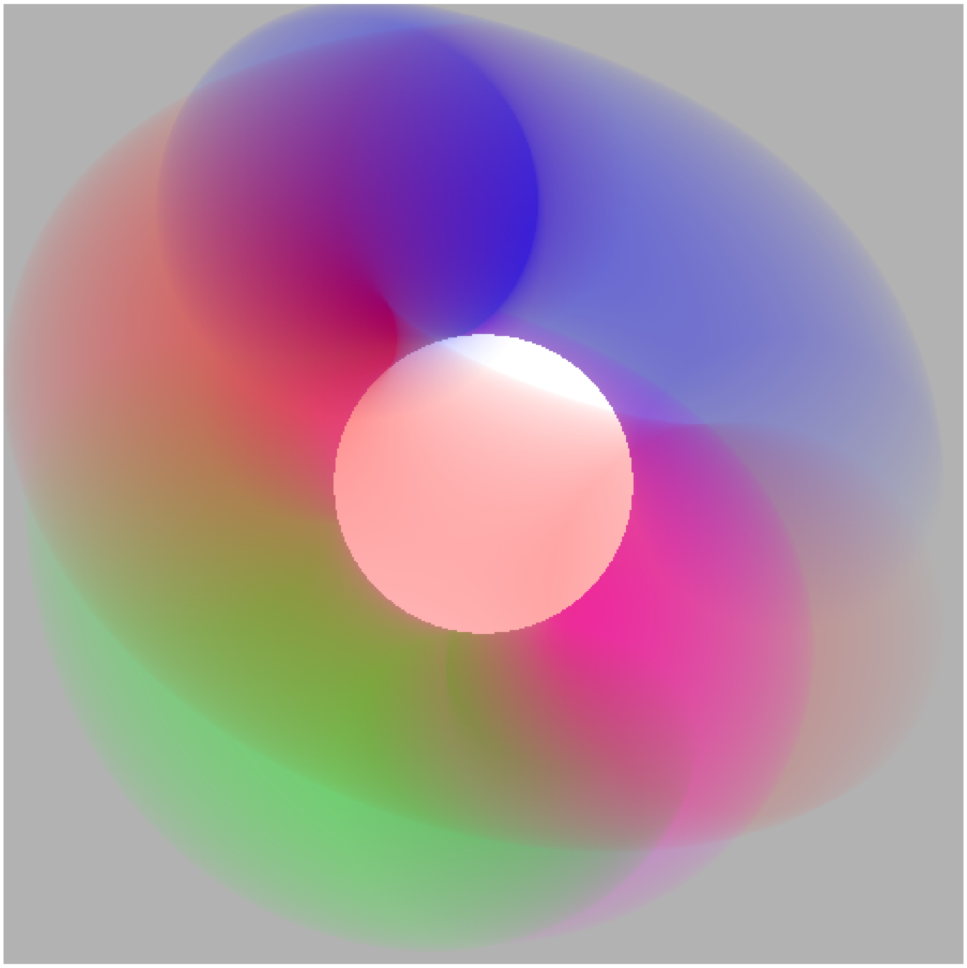
4.5 More Results

To further verify the usefulness of the MDF3 algorithm, we consider a more complicated test case, illustrated in Figure 4.21. The scene consists of 4 coloured spheres attached to a central white sphere. The white sphere rotates down and to the right over the exposure time interval, causing the coloured spheres to sweep around it, as shown in the snapshots in the figure.

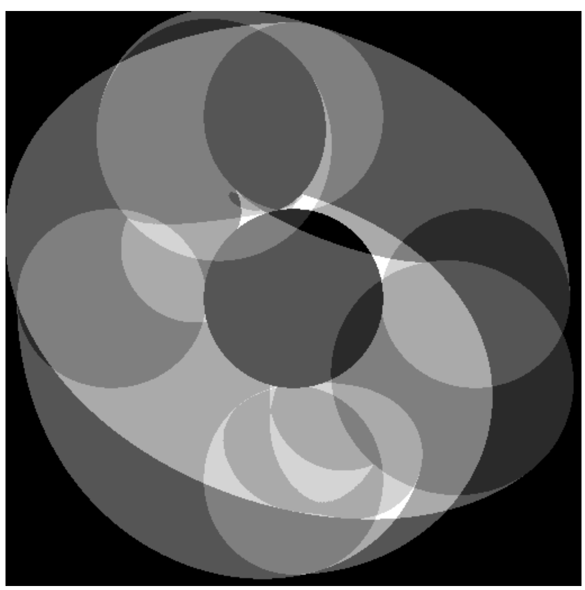
Figure 4.22 shows exact motion-blurred and discontinuity images of the scene, and Figures 4.23 and 4.24 show the results produced by MDF3 and Stratified Random Sampling. Tolerances of $\epsilon = 0.02$ and $\nu = 0.2$ were used for MDF3, resulting in an average of 16.09 visibility samples per pixel. A cache size of 25 pixels was used, and the cost of backtracking was 0.08 samples/pixel, on average. At this quality setting, the result is almost indistinguishable from the exact solution. At 16 samples per pixel, the Stratified Random Sampling result still displays considerable noise and noticeable banding artifacts.

4.6 Conclusions

In this chapter, we applied the ideas of the TDF algorithm to the case of an arbitrary number of visibility discontinuities. We formulated the MDF1, MDF2, and MDF3 visibility sampling algorithms, which exploit differing degrees of pixel–to–pixel coherence in the visibility function. By propagating coherence information in both directions along each scanline, the final MDF3 algorithm is capable of producing visibility solutions that are noticeably smoother than those produced by Stratified Random Sampling at equivalent sample counts. In the next chapter, we explore algorithms for computing the shading integrals of non–constant–coloured objects based on the superior visibility solution provided by MDF3.

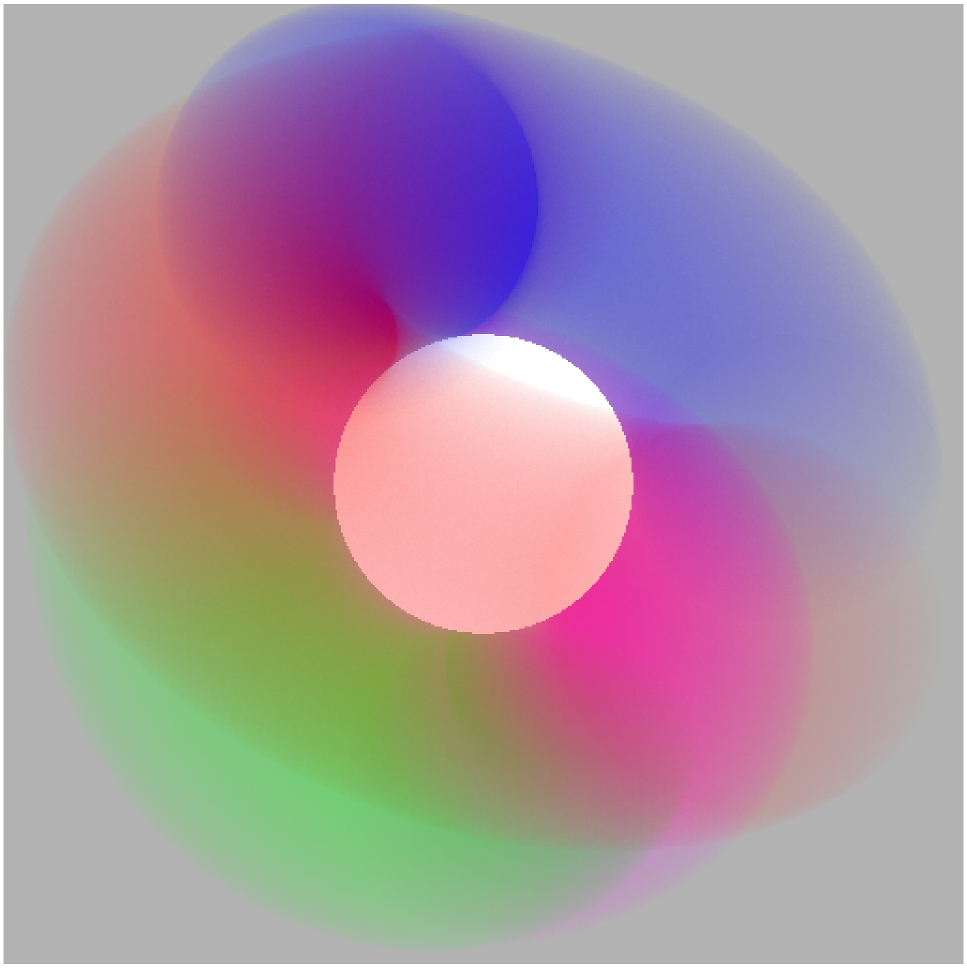


Motion-Blurred Image

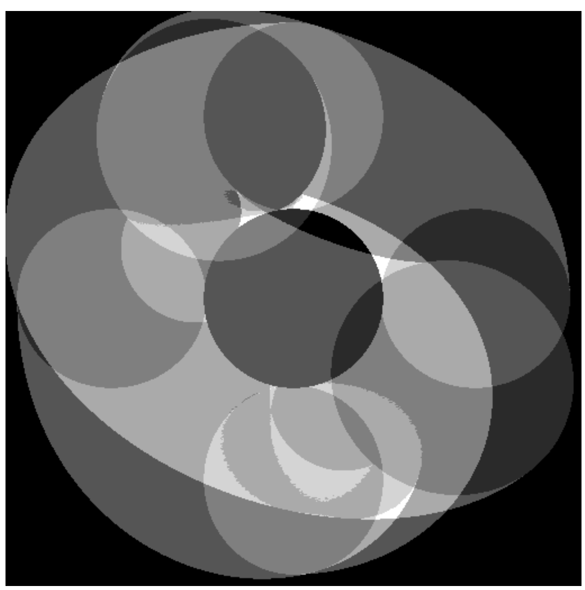


Discontinuity Image

Figure 4.22: Exact results.

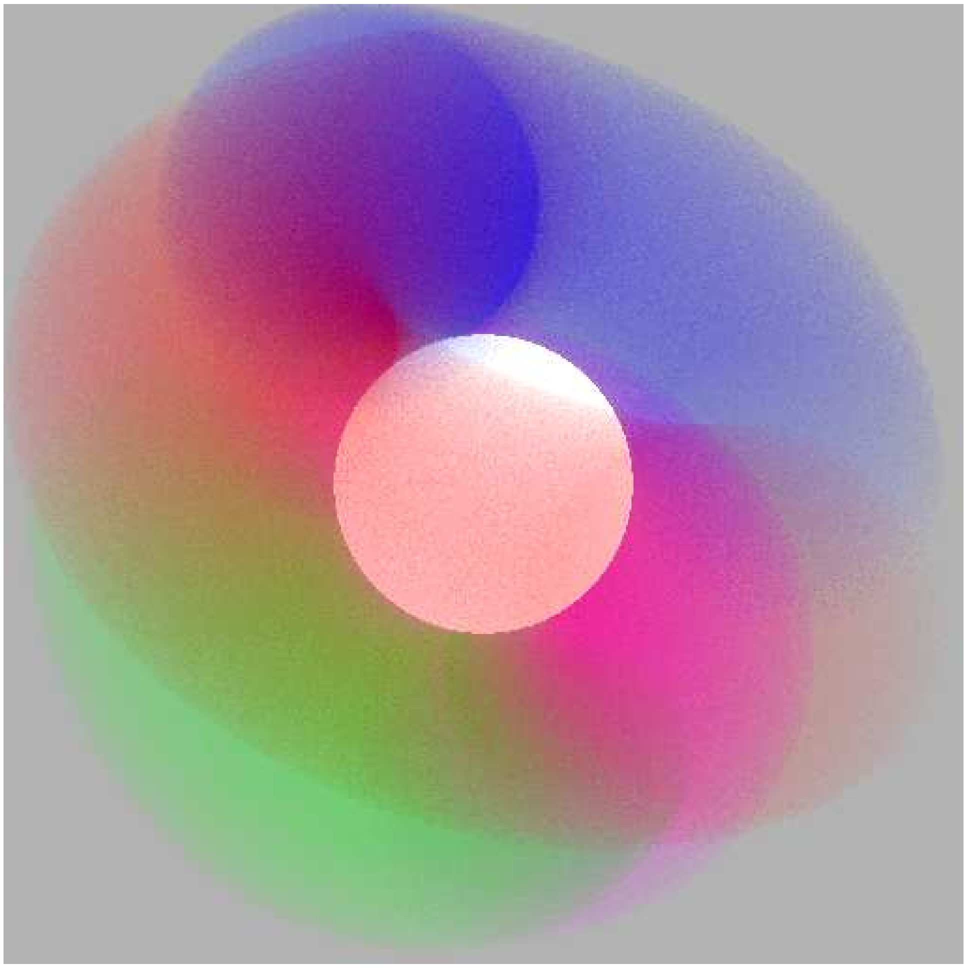


Motion-Blurred Image

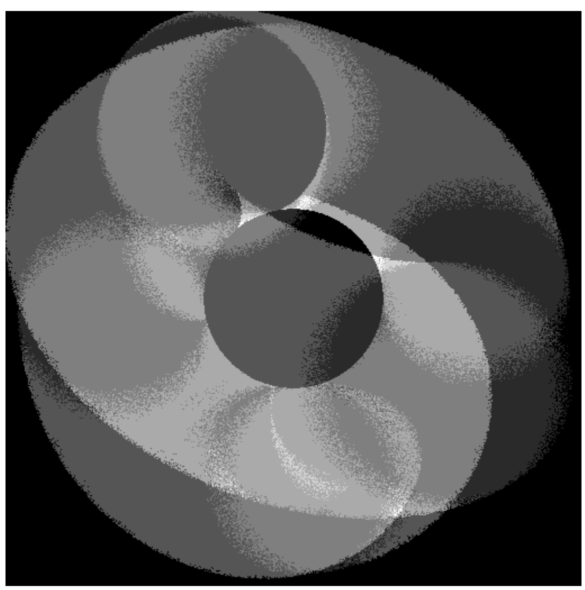


Discontinuity Image

Figure 4.23: MDF3 results ($\epsilon = 0.02, \nu = 0.2$): 16.09 samples/pixel



Motion-Blurred Image



Discontinuity Image

Figure 4.24: Stratified Random Sampling results: 16 samples/pixel

Chapter 5

Temporal Shading

5.1 Introduction

In Chapter 3, we formulated the motion blur integral as follows:

$$I_{xy} = \sum_{i=1}^{N} \int_{T_i} h_t(t_{mid} - t) I_i(t) \, dt \tag{5.1}$$

Each term in the summation represents a particular object's contribution to the final pixel colour. This contribution is simply the integral of the object's intensity function (or *shading function*) $I_i(t)$ over the time interval T_i during which it is visible, weighted by the temporal filter function h_t. We refer to this integral as the object's *shading integral*. While previous chapters have focused on determining the domains of integration T_i, we will now concentrate on the problem of actually evaluating the shading integrals over these domains.

Our approach will be to approximate the shading integrals using point sampling. Each object's intensity function will be sampled at a number of times within its visible interval, and the results of these *shading samples* combined to produce our shading integral approximation for the object. We will develop a sampling strategy called *Object-Based Shading* (OBS) for controlling the distribution of these shading samples. Under OBS, each object is sampled independently using a sampling pattern chosen based on the speed of the object and the complexity of its surface appearance. This strategy allows us to obtain good shading integral approximations using as few shading samples as possible.

Armed with the MDF3 and OBS sampling strategies, we can describe a two-stage algorithm for computing motion-blurred images using Equation 5.1. In the first stage, visibility is sampled using MDF3 to compute the visible intervals T_i. The second stage then applies OBS to sample the shading functions over these visible intervals. We refer to this composite approach as the *decoupled MDF3/OBS* algorithm, where the term "decoupled" refers to the fact that visibility and shading are sampled independently using different sampling strategies. By contrast, Distribution Ray Tracing is based on a *coupled SRS* (Stratified Random Sampling) algorithm in which visibility and shading are both sampled using a single stratified random sampling pattern. We will compare the results of our approach to those of Distribution Ray Tracing, which is currently the most popular point–sampling approach to generating motion blur.

In Section 5.2, we describe the Object–Based Shading algorithm. In Section 5.3, we explore the integration of shading and visibility sampling strategies, describing why we expect our decoupled MDF3/OBS algorithm to prove superior to the coupled SRS approach of Distribution Ray Tracing. Section 5.4 describes the implementation of the coupled and decoupled approaches in our renderer, and Section 5.5 compares our results to those of Distribution Ray Tracing on several different scenes. Finally, conclusions are presented in Section 5.6

5.2 Object–Based Shading

In Equation 5.1, each object's shading integral is independent, so we are free to use different sampling strategies for each. We wish to tune the sampling of each object to produce a good estimate of its shading integral using as few samples as possible. For each visible interval, we must make two choices: how many shading samples to use, and where in the interval those samples should be placed. The following sections describe how each of these choices are made by the Object–Based Shading algorithm.

5.2.1 Choosing the Number of Samples

The number of samples required to accurately approximate a particular shading integral depends on the complexity of the temporal shading function $I_i(t)$. The more complicated the shading function, the more shading samples necessary to capture its behaviour. The complexity of an object's temporal shading function is influenced by two main factors: the speed of the object and the complexity of its surface appearance. In Figure 5.1, we see the temporal shading function resulting from three objects of differing surface appearances passing through a pixel at different speeds.

Object 1 has a simple surface appearance (a constant colour), resulting in a constant-valued temporal shading function. As the object passes through the pixel, the object colour observed through the pixel centre remains constant. For such a simple shading function, a single shading sample captures all the

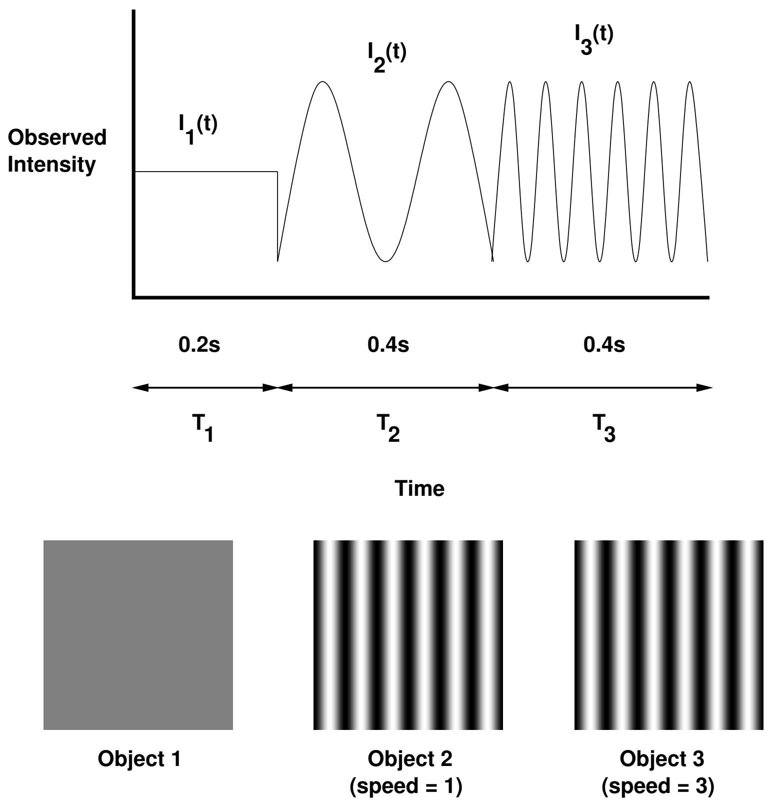


Figure 5.1: Temporal shading function resulting from the motion of three objects through a pixel.

relevant information about the function. For Object 2, the wave-like temporal shading function is the result of two peaks of the undulating surface pattern passing under the pixel centre as the object moves. For this more complicated case, a minimum of 4 shading samples are required to capture the relevant features of the function. Object 3 has the same surface appearance as Object 2, but moves at three times the speed, resulting in three times as many undulations in the shading function. This more complicated shading function requires at least 12 shading samples to capture all of its peaks and valleys.

This example demonstrates that different objects have different shading sampling requirements. Slow moving objects with simple appearances may require only a few shading samples, while more complicated or fast moving objects may require many samples. Object–Based Shading attempts to take advantage of this fact. The user assigns to each object i a *basic shading sampling rate* s_i based on its speed and the complexity of its surface appearance. For each visible interval T in which object i is visible, the number of shading samples allocated is $\lceil s_i\, a(T) \rceil$, where $a(T)$ is the area of the visible interval T under the temporal filter function $h_t(t)$. The area term is included to ensure that the sampling effort devoted to each interval is proportional to its expected contribution to the final pixel colour, and the ceiling function is used to ensure that each visible interval receives at least one shading sample. The basic shading sampling rate is stated in units of samples/second, since it is a measure of temporal sampling density (we assume for the sake of simplicity that the exposure time interval is always exactly 1 second long). Given the interval widths in the figure, basic shading rates of 1, 10, and 30 samples/second in our example would result in approximately the right number of samples being devoted to objects 1, 2, and 3 respectively. Selection of basic shading rates is not an exact science, since the exact form of the shading function is unknown prior to sampling. The best we can do is make a rough guess based on object speed and surface appearance.

5.2.2 Choosing the Sampling Pattern

Since the exact form of the shading function is unknown prior to sampling, important features of the shading function could be located in any area of an object's visible interval. Once we have chosen how many shading samples to cast, the best way to distribute shading samples is to spread them evenly over the visible interval in hopes of detecting all relevant features, wherever they may lie. In this thesis, we have seen two sampling strategies that spread samples evenly, Uniform Sampling and Stratified Random Sampling. In Object–Based Shading, each object in the scene is assigned one of these sampling patterns by the user. As with the basic shading rate, selecting an appropriate sampling pattern can be tricky, but there are certain guidelines that can be followed.

If the temporal shading function is relatively smooth, then deterministic sampling patterns such as

Uniform Sampling can produce good shading integral approximations, assuming that the sampling rate is high enough. For non-smooth shading functions (those containing sharp edges, for example), however, Uniform Sampling will tend to produce coherent, structured artifacts (such as sharp colour bands) in the resulting images. These artifacts are the result of high-frequency components in the shading function aliasing as structured low-frequency information in the integrated result. In such cases, randomized approaches such as Stratified Random Sampling can be used to alleviate the problem. Presented with high-frequency information, randomized approaches will still generate artifacts, but the artifacts will take the form of image noise, which most observers find more tolerable than the structured artifacts resulting from Uniform Sampling.

5.2.3 Putting Them Together

In Object–Based Shading, the user specifies for each object in the scene a basic shading sampling rate and a shading sampling pattern (Uniform or Stratified Random). These parameters determine how that object's visible intervals are sampled. For each interval, we compute the number of shading samples n, then stratify the interval into n strata of equal area under the temporal filter function $h_t(t)$. Each of these strata receives a single shading sample. If the object is to be shaded using Uniform Sampling, each sample is placed at the midpoint (with respect to area under the filter function) of its stratum. If Stratified Random Sampling is being used, the temporal filter function is used to place each sample randomly within its stratum.

Figure 5.2 shows an example of an OBS shading sampling pattern for a pixel in which two objects are visible. Object 1 uses Stratified Random Sampling with a basic shading sampling rate of 8 samples/second, while Object 2 uses Uniform Sampling with a basic shading rate of 4 samples/second. Both visible intervals have area 0.5 under the Gaussian temporal filter, so interval T_1 receives 4 samples, and T_2 receives 2 samples. Each interval's shading integral is then simply the average of the colours computed at these sample times. Assuming that the resulting sampled colours are $\{C_1, \ldots, C_6\}$, then the final pixel colour is $0.5 * \frac{C_1+C_2+C_3+C_4}{4} + 0.5 * \frac{C_5+C_6}{2}$.

5.3 Advantages of Decoupled MDF3/OBS Sampling

Computation of pixel colours using Equation 5.1 requires sampling of both visibility and shading. In the decoupled MDF3/OBS algorithm, these functions are sampled independently using different sampling strategies (MDF3 for visibility, OBS for shading). In the coupled SRS algorithm used by Distribution Ray Tracing, a single stratified random sampling pattern is used to sample both visibility and shading. In this section, we

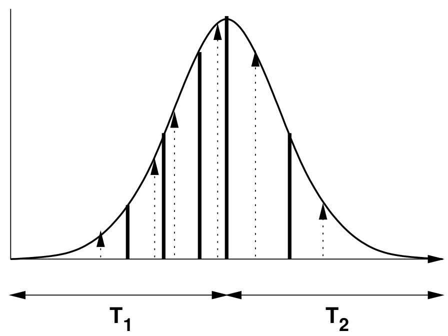


Figure 5.2: Object–Based Shading sampling pattern. Each interval's samples are distributed by stratifying the interval under the Gaussian temporal filter function shown. The dark lines represent the shading strata, and the dotted arrows show the actual sample positions within the strata. Object 1 uses Stratified Random Sampling, so its samples are randomly placed within their strata. Object 2 uses Uniform Sampling, so its samples are placed at the centres of their strata.

explain why we expect that our decoupled algorithm will prove superior to the coupled algorithm.

The main benefit of the decoupled approach derives from the fact that Equation 5.1 expresses the final pixel colour in terms of several different functions, namely the visibility function and the shading functions of each object. All of these functions may have vastly different characteristics, and thus should be sampled using different strategies. For example, the visibility function is a step function whose only significant features are its discontinuities, so it makes sense to sample it using an adaptive strategy that concentrates solely on locating these discrete features. By contrast, shading functions generally exhibit more continuous variation, so the best way to sample them is to spread shading samples evenly over their domains. Furthermore, the previous section showed that shading functions can vary dramatically amongst objects, so different objects may have different shading sampling requirements.

Attempting to appease all these different components of the motion blur equation simultaneously (and efficiently) is a difficult task. The visibility function tells us to concentrate samples around the visibility discontinuities, while the shading functions tell us to spread samples evenly throughout each visible interval (with some intervals demanding more samples than others, of course!). In the coupled SRS approach, the user is forced to select a single stratified random sampling pattern that *simultaneously satisfies all these criteria*. Typically the only way to achieve this is to choose a sampling density that is high enough to satisfy the most demanding component (usually the visibility function). For example, 100 SRS visibility samples may be required to eliminate banding artifacts caused by visibility discontinuities. Since visibility and shading sampling are coupled, however, the shading function of the visible object must also be sampled at each of these 100 sample times (which will be spread roughly evenly over the exposure time interval). Often this will result in gross oversampling of the shading functions of one or more objects. Similarly, if more shading samples are required (for a particularly troublesome object, say), then the number of visibility samples must also be increased, possibly resulting in oversampling of the visibility function. Thus, the coupling of visibility and shading sampling result in an often wasteful (and hence inefficient) algorithm.

By decoupling visibility and shading sampling, the decoupled MDF3/OBS algorithm avoids this problem. Visibility is sampled first, using MDF3 to compute a high-quality visibility approximation much more efficiently than SRS. OBS then independently samples the resulting shading integrals, using only as many shading samples as it feels are necessary for each object. With this algorithm, each component of Equation 5.1 (the visibility function and each object's shading function) is sampled independently using an appropriate strategy. Rather than being a slave to the most demanding component in the equation, the decoupled MDF3/OBS algorithm is able to allocate samples only where they are needed. For this reason, we expect that this algorithm will prove more efficient than the coupled SRS algorithm.

5.4 Implementation

In order to understand the implications of selecting coupled SRS or decoupled MDF3/OBS sampling, we will now describe our renderer's implementation of both approaches. First, we describe the general issues involved in computing shading and visibility, then we detail the specifics of our algorithms.

5.4.1 Shading

In our renderer, each object has an associated *surface shader* that determines its appearance. The surface shader is simply a function that computes the colour of a point on the surface, based on geometric and lighting information about the surface. Shaders can be extremely complex, sometimes consisting of thousands of lines of code that implement complicated texturing and lighting effects. These shaders will often require detailed geometric information (called *sample information*) about the point being shaded. For example, lighting models need to know the surface normal and direction to the viewer at the point being shaded, as well as the position and colour of every light source in the scene. Texture mapping algorithms often need to know the size and shape of a screenspace filter projected onto the point being shaded, for the purposes of antialiasing the texture. All this sample information must be computed by the renderer and passed as parameters to the shader. Generation of sample information and execution of surface shaders can be a very expensive proposition, so reducing the number of shading samples can have a large impact on total rendering time.

```
colour ComputePixelColour_Coupled_SRS()
{
    Compute visibility/shading sample times{t_1, ..., t_n}
        using Stratified Random Sampling

    foreach sample time t_i {
        Move all objects in the scene to their positions at time t_i
        Cast a ray through the pixel centre to determine the
            nearest visible object k
        Compute sample information at hit point P_k on object k
        Invoke object k's surface shader to compute colour at P_k
    }

    Return average of all computed colours
}
```

Figure 5.3: Coupled SRS algorithm.

5.4.2 Visibility

In order to implement the visibility sampling schemes that we have discussed in this thesis, our renderer needs to be able to determine the object that is visible at a particular time through a particular pixel centre. To accomplish this, we use a spacetime raytracing algorithm. To cast a visibility sample, we first move all objects in the scene to their positions at the visibility sample time, then we trace a ray into the scene through the pixel centre. The ray is intersected with each object in the scene, and the intersection point nearest to the viewer (the "hit point") is determined. Sample information can then be computed for this hit point.

Updating the position of every object in the scene every time a ray is cast can get quite expensive, so our renderer uses several levels of spacetime bounding boxes to ensure that only objects that could potentially intersect the ray are updated. Still, visibility sampling of a dynamic scene is expensive, so we would like to cast as few visibility samples as possible.

5.4.3 Shading/Visibility Interaction

In this section, we describe how the shading and visibility components of our renderer interact in our implementations of the coupled SRS and decoupled MDF3/OBS sampling algorithms. Figures 5.3 and 5.4 give pseudocode for the coupled and decoupled algorithms, repectively. In the coupled SRS approach, a set of sample times is selected using Stratified Random Sampling. For each of these sample times, we compute the "hit point", by intersecting the ray with every potentially visible object in the scene. Sample data is

```
colour ComputePixelColor_Decoupled_MDF3_OBS()
{
    // Visibility Stage (estimate visible intervals using MDF3)
    foreach visibility sample time v_i required by MDF3 {
        Move all objects in the scene to their positions at time v_i
        Cast a ray through the pixel centre to determine the
            nearest visible object
    }

    // Shading Stage (compute shading integrals using OBS)
    foreach visible interval T_j computed by MDF3 {
        Let O_j be the visible object for this interval
        Compute number of shading samples n for the interval
        Compute shading sample times {s_1, ..., s_n}

        foreach shading sample time s_k {
            Move object O_j to its position at time s_k
            Cast a ray through the pixel centre to determine the
                hit point P_k on the object
            Compute sample information at hit point P_k
            Invoke object O_j's surface shader to compute colour at P_k
        }

        Visible interval T_j's contribution to the final pixel color
            is the average of these colours weighted by the area of T_j
            under the filter function
    }

    Return sum of all computed interval colours
}
```

Figure 5.4: Decoupled MDF3/OBS algorithm.

then computed for the hit point, and the object's surface shader is invoked to determine the sample colour. The final pixel colour is simply the average of all the computed sample colours.

In the decoupled MDF3/OBS approach, visibility is handled first, with MDF3 determining the visibility sample times. For each of these visibility samples, we compute the hit point by intersecting with all potentially visible objects in the scene. Since we are only concerned with determining the visible object, we do not need to compute sample data or invoke the object's surface shader. Once the visible intervals for the pixel have been computed, we then move on to the shading stage. For each visible interval, we select a set of shading sample times using OBS. For each of these shading sample times, we know what object is visible, but we do not know exactly which point on the object is visible. For this reason, we must cast **another**

visibility ray to compute the hit point on the object, from which sample data is computed and passed to the surface shader. The resulting colours are averaged and weighted by the area of the visible interval under the filter function to produce the interval's contribution to the final pixel colour.

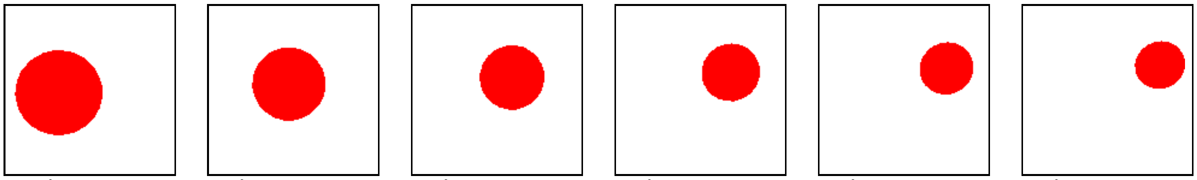
It is unfortunate that each shading sample requires us to cast another visibility ray at the object being shaded, but we should keep in mind that these visibility samples are not as expensive as those cast in the visibility stage of the algorithm. Whereas the earlier samples must consider every object in the scene as a potential intersector, we know in advance which object the shading visibility ray will hit. All other objects in the scene are ignored when these rays are cast, so their cost is lessened.

5.5 Results

Having described our algorithms, we now examine their performance under a variety of shading conditions. Our first test scene consists of a single sphere moving diagonally away from the camera. There are three point light sources in the scene - above, below, and to the right of the camera. We will explore three increasingly complicated lighting models (ambient, diffuse, diffuse–plus–specular), as well as texture mapping. In each case, the paramters of the coupled and decoupled algorithms were set to yield the same total render time for each algorithm. All images were rendered at a resolution of 400x400 pixels, and all tests were performed using our *RenderDude!* renderer running under Linux on a Pentium II (266 MHz) machine with 128 MB of RAM.

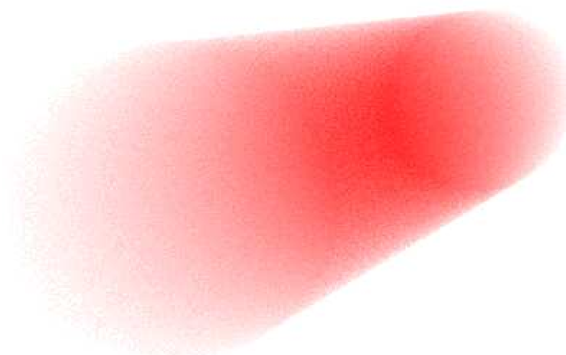
5.5.1 Ambient Lighting

Illumination models used in graphics typically incorporate an "ambient" term to simulate indirect lighting. Usually, this term is simply a constant colour. In this example, we apply an ambient lighting model to the moving sphere. The "ambient colour" of the object is red, so the object's surface shader returns the colour value $(R, G, B) = (1, 0, 0)$ for every point on the object at every time. Figure 5.5 shows several snapshots of the moving sphere, and images produced using the coupled and decoupled algorithms. The figure shows that the decoupled approach produces a superior image, free of the banding and noise artifacts visible in the coupled image. This should not be surprising, since in this case, visibility is the sole cause of variability in the pixel colour, so we would expect the strategy employing the superior visibility algorithm (MDF3) to win out.

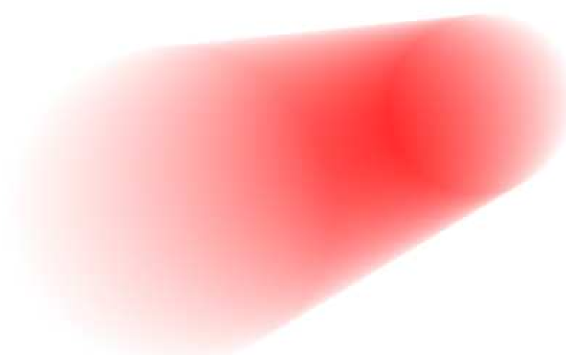


Time = 0 **Time = 0.2** **Time = 0.4** **Time = 0.6** **Time = 0.8** **Time = 1.0**

Snapshots

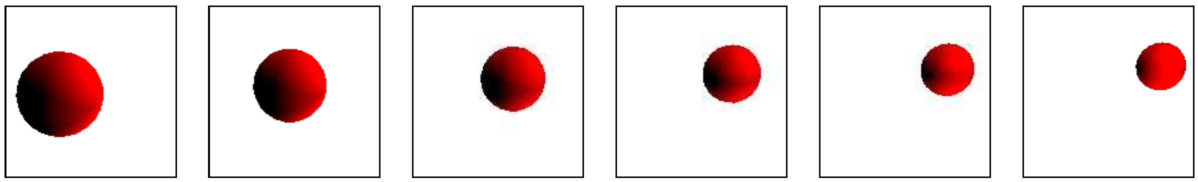


Coupled SRS: 16 samples/pixel



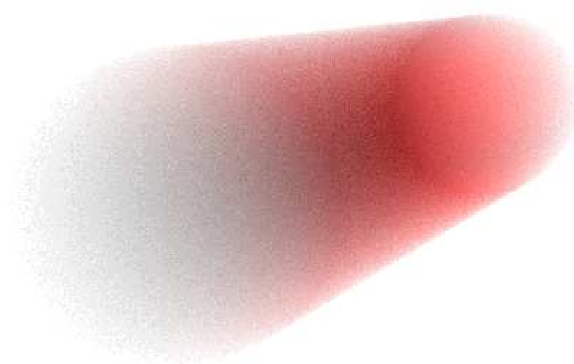
Decoupled MDF3/OBS: Uniform shading sampling, 1 sample/second basic shading rate

Figure 5.5: Ambient lighting: For this simple constant–shaded case, the decoupled approach used MDF3 visibility settings of $\epsilon = 0.01$, $\nu = 0.2$ with a cache size of 25 pixels, resulting in an average of 11.5 visibility samples/pixel. Each image took 29 seconds to render.

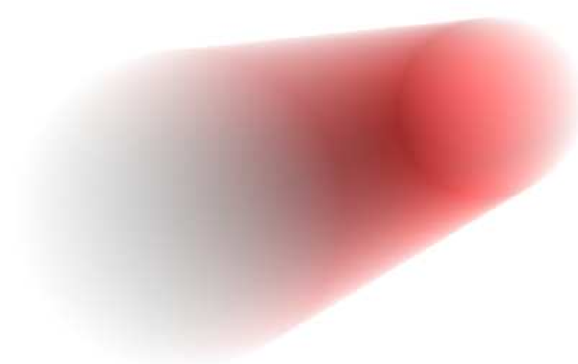


Time = 0 **Time = 0.2** **Time = 0.4** **Time = 0.6** **Time = 0.8** **Time = 1.0**

Snapshots



Coupled SRS: 17 samples/pixel



Decoupled MDF3/OBS: Uniform shading sampling, 10 samples/second basic shading rate

Figure 5.6: Diffuse lighting: For this slightly more complicated lighting model, the decoupled approach uses a basic shading rate of 10 samples/second, resulting in an average of 3.4 shading samples for each pixel in which the object is visible. The decoupled approach uses the same visibility settings as in the ambient lighting case. Each image took 36 seconds to render.

5.5.2 Diffuse Lighting

For our next example, we apply a Lambertian shader to the object. The Lambertian model simulates the "diffuse" shading resulting from the reflection of direct illumination from a matte surface. The intensity I of the light diffusely reflected from a surface point P is modelled by the following formula:

$$I = \sum_{i=1}^{L} k_d \, I_i \, max(0, N \cdot L_i) \tag{5.2}$$

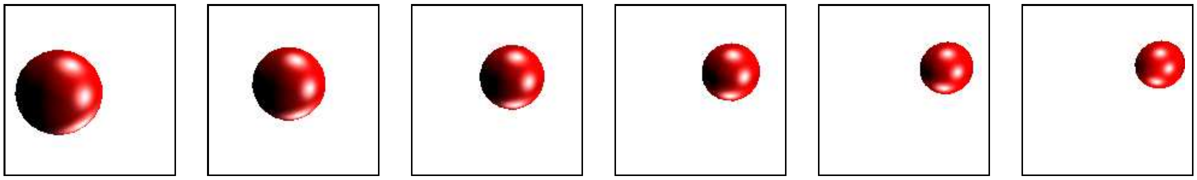
where k_d is the "diffuse colour" of the object, I_i is the intensity of the i^{th} light source in the scene, L_i is the direction from P to that light source, and N is the surface normal at P. In our scene, the object's diffuse colour is red, and the surface is illuminated by three white lights, located above, below, and to the right of the viewer.

Figure 5.6 shows snapshots of the object's appearance over the exposure time interval, as well as images generated using the coupled and decoupled approaches. For the decoupled approach, we used the same visibility settings as in the previous example, resulting in an average of 11.5 visibility samples/pixel being cast. The diffuse lighting function is sufficiently smooth that all pixels in the image observe a smooth variation in the object's temporal shading function $I_i(t)$, so we use Uniform Sampling to shade the sphere. In order to capture the relatively minor variations in the shading function over time, we turned the object's basic shading rate up to 10 samples/second, resulting in an average of 3.4 shading samples per pixel. The total render time for the decoupled approach was 36 seconds. We applied the coupled approach using 17 samples per pixel, which also took 36 seconds to render the image.

The image produced by the decoupled algorithm is clearly superior to that of the coupled algorithm. The decoupled image exhibits none of the noise or banding artifacts apparent in the coupled image, even though the coupled approach takes on average 70% more shading samples per pixel. By spending more time sampling the discontinuous visibility function and less time sampling the relatively simple shading function, the decoupled approach performs much better in this case.

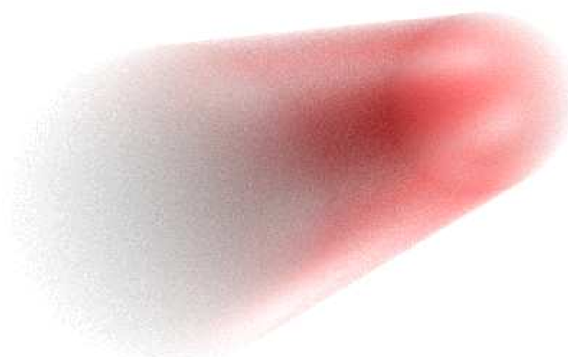
5.5.3 Diffuse–plus–Specular Lighting

Another effect commonly simulated in lighting models is the mirror reflection (known as a "specular highlight") of a light source as seen in extremely smooth materials. A wide range of materials can be simulated by combining this specular model with the diffuse model from the previous section. The resulting lighting

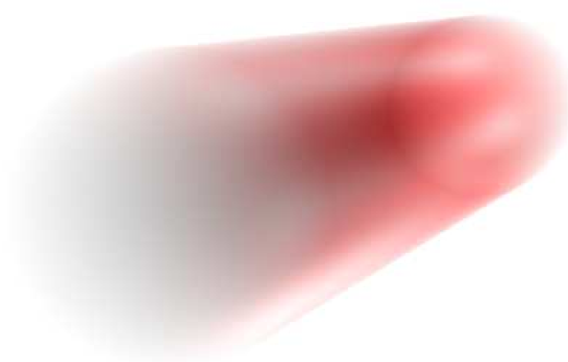


Time = 0 **Time = 0.2** **Time = 0.4** **Time = 0.6** **Time = 0.8** **Time = 1.0**

Snapshots



Coupled SRS: 19 samples/pixel



Decoupled MDF3/OBS: Uniform shading sampling, 20 samples/second basic shading rate

Figure 5.7: Diffuse–plus–Specular lighting: The more complex variation introduced by the specular highlights requires an increase in the basic shading rate to 20 samples/second. This results in an average of 6.8 shading samples being cast for every pixel in which the object is visible. The total render time for each algorithm is 41 seconds.

model can be expressed in the following equation:

$$I = \sum_{i=1}^{L} (k_d \, I_i \, max(0, N \cdot L_i) + k_s \, I_i \, max(0, V \cdot R_i)^n) \tag{5.3}$$

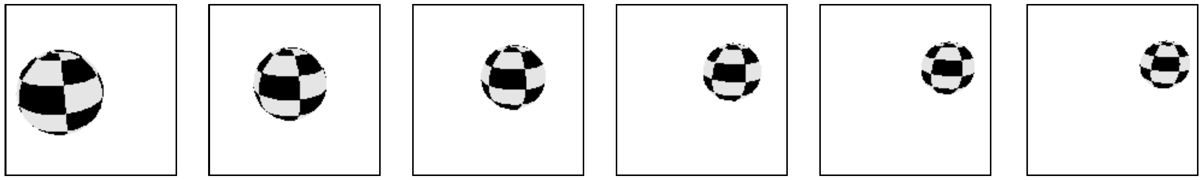
where k_s is the "specular colour" of the object, V is the direction to the viewer from the point P being shaded, R_i is the mirror reflection direction for the i^{th} light source, and the specular exponent n controls the size of the highlight. Figure 5.7 depicts our now-familiar sphere scene using this new lighting model. We use the same diffuse lighting parameters as before, and we set the specular colour of the object to be white. Since there are three white lights in the scene, we observe three white specular highlights that move across the object with time.

The specular highlights complicate the object's temporal shading function, but it still remains fairly smooth, so we continue to use Uniform Sampling for the shading of the object, and turn the basic shading rate for the object up to 20 samples/second. Using the same visibility settings as before, the resulting decoupled render took 41 seconds, the same as the coupled algorithm using 19 samples per pixel. With these settings, both algorithms cast roughly the same number of shading samples in each pixel, but the decoupled approach produces a less noisy image because of its superior visibility estimate.

5.5.4 Texture Mapping

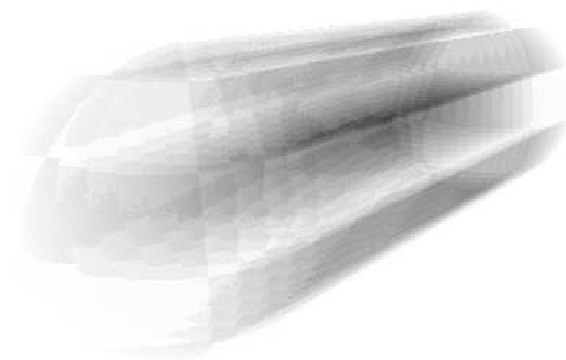
The lighting models presented thus far are all reasonably well-behaved, in that they do not introduce extremely high frequencies into the object's temporal shading function $I_i(t)$. For this reason, Uniform Sampling did a good job on our lighting test cases, producing images free of any noticeable artifacts. In general, however, surface shaders can make use of much more complicated functions to describe an object's appearance. One commonly used technique is *texture mapping*, in which an image is pasted onto the surface of the object. If the texture image contains high-frequency information (sharp edges are a common example), then Uniform Sampling will produce coherent banding artifacts. As an example, consider a simple black-and-grey checker texture applied to our moving sphere, as shown in Figure 5.8. If we use the same shading settings as in the specular shading case (Uniform Sampling with a basic shading rate of 20 samples/second), the result is the top image in the figure, which displays disturbing banding artifacts. In cases such as this, we should switch to Stratified Random Sampling for our shading strategy. The lower image in the figure shows that using Stratified Random Sampling at the same basic shading rate replaces the coherent banding artifacts with noise, which is much less objectionable.

The shading problem presented by edges in a moving texture map is reminiscent of the temporal visi-

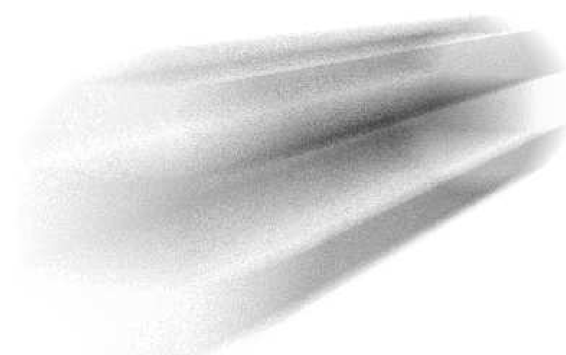


Time = 0 **Time = 0.2** **Time = 0.4** **Time = 0.6** **Time = 0.8** **Time = 1.0**

Snapshots

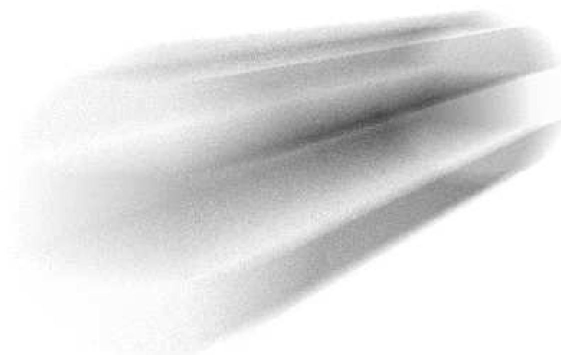


Decoupled MDF3/OBS: Uniform shading sampling, 20 samples/second basic shading rate



Decoupled MDF3/OBS: Stratified random shading sampling, 20 samples/second basic shading rate

Figure 5.8: Texture mapping: These images demonstrate that Uniform Sampling is not appropriate for shading functions containing high-frequency information. The banding artifacts present in the Uniform Sampling image are transformed into noise when we switch to Stratified Random Sampling for shading.



Coupled SRS: 30 samples/pixel

Decoupled MDF3/OBS: Stratified random shading sampling, 80 samples/second basic shading rate

Figure 5.9: Texture mapping: In order to generate a smooth image, the checkered sphere requires the use of Stratified Random Sampling for shading, and a basic shading rate of 80 samples/second (which results in an average of 26.2 shading samples for every pixel in which the object is visible). For the same amount of rendering time (55 seconds), the 30-sample coupled algorithm produces an image that still contains some noise.

bility problem discussed in previous chapters - both problems involve integrating a discontinuous function. Although our earlier investigations showed that superior results can be achieved in such cases by using adaptive sampling to actively seek out the discontinuities, we opt for the non-adaptive strategy for the shading problem because it is simple, and it can handle both smooth and discontinuous shading functions. In the presence of discontinuities, we simply turn up the basic shading rate to reduce the impact of noise and banding artifacts on the final image. In Figure 5.9, we use a basic shading rate of 80 samples/second in our decoupled algorithm to produce a reasonably smooth image in 55 seconds of rendering time. In the same amount of time, the image produced by the coupled algorithm (using 30 samples per pixel) still exhibits some noise. Although its benefits are less pronounced for discontinuous shading functions, the decoupled approach is still the better choice.

5.5.5 Putting Them All Together

The benefits of decoupled shading/visibility become especially apparent in a scene containing several objects with widely differing shading requirements. Figure 5.10 shows snapshots of such a scene, containing five spheres that spin counterclockwise as they move away from the camera. Each sphere uses one of the surface shaders described in the previous sections. The centre sphere uses a flat red shader, the lower right sphere uses a grey diffuse shader, the top right sphere uses a primarily specular shader, the top left sphere uses a checker texture map, and the bottom left sphere uses a combined diffuse/specular model. The scene contains white, blue, and red lights.

The decoupled algorithm was used to produce the image in Figure 5.11, using basic shading rates of 1 sample/second for the flat-shaded red sphere, 10 samples/second for the diffuse object, 20 samples/second for the two specular objects, and 100 samples/second for the checker textured object. Uniform sampling was used for shading all objects except the checkered sphere, which used Stratified Random Sampling. These settings are sufficient to produce a smoothly blurred image. In order to produce the same quality result using the coupled approach, our sampling rate is constrained by the most demanding object in the scene, namely the checkered sphere. In order to guarantee that the checkered sphere receives the same number of shading samples as in the decoupled algorithm, we are forced to use a global sampling rate of 100 shading/visibility samples per pixel in the coupled algorithm. This rate does a good job on the checkered sphere, but it is overkill for all the other objects in the scene. The result is that the coupled algorithm takes 480 seconds of rendering time to produce an image (shown in Figure 5.12) of the same quality as was produced by the decoupled algorithm in just 108 seconds.

All the images presented thus far have been generated using temporal box filters. Figure 5.13 shows an

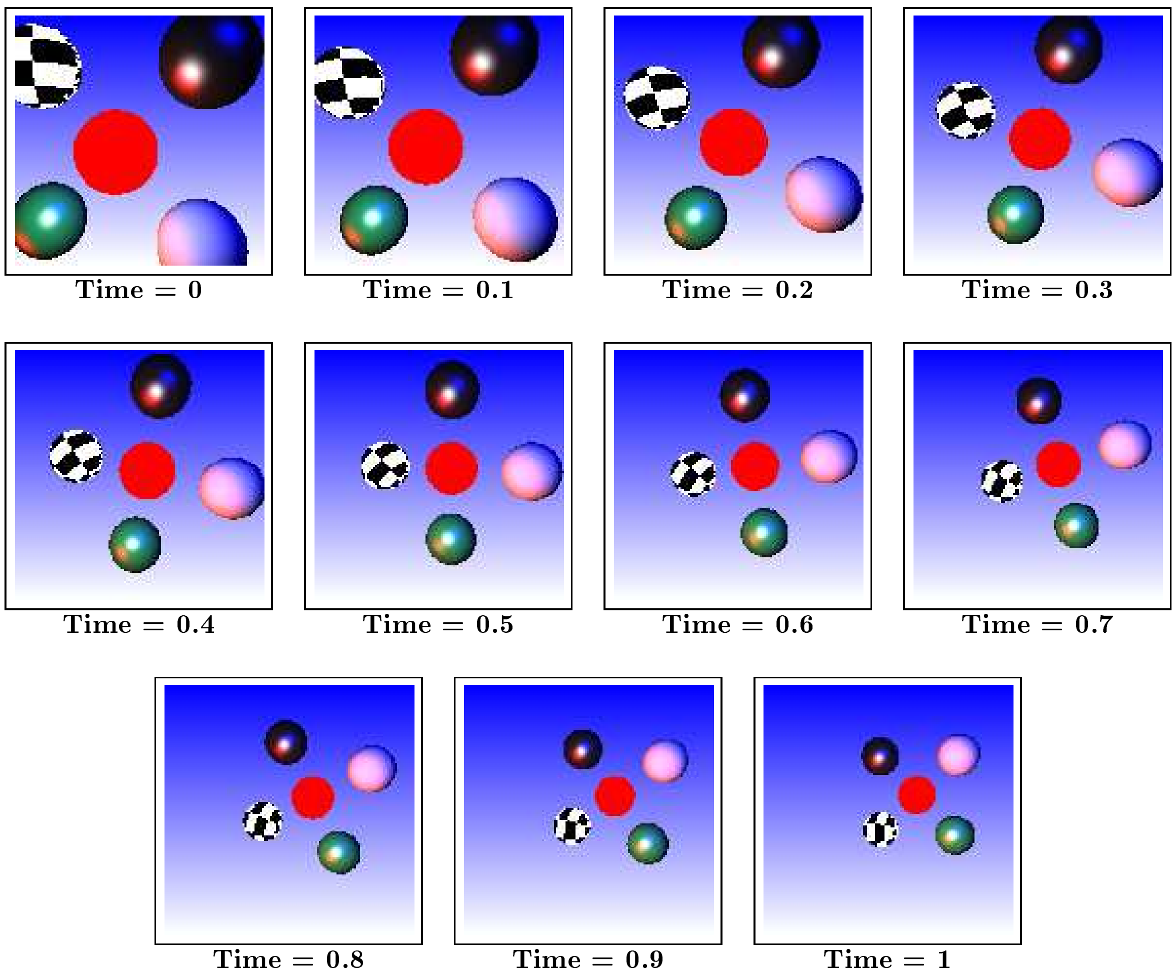


Figure 5.10: Combination scene snapshots.

image produced by the decoupled algorithm using a temporal Gaussian filter, to illustrate that our algorithm handles this case.

5.6 Conclusions

In this chapter, we have observed the benefits of decoupling the shading problem from the visibility problem in synthesizing motion-blurred images. The decoupled MDF3/OBS sampling algorithm simultaneously eliminates image artifacts resulting from the discontinuous visibility function and allows individual control over the shading sampling of each object. As a result, the decoupled approach can produce high-quality motion-blurred images in less time than is required by the coupled SRS approach, leading us to conclude that decoupled MDF3/OBS is the better algorithm.

Figure 5.11: Combination scene (Decoupled MDF3/OBS - box filter): 1:48 rendering time

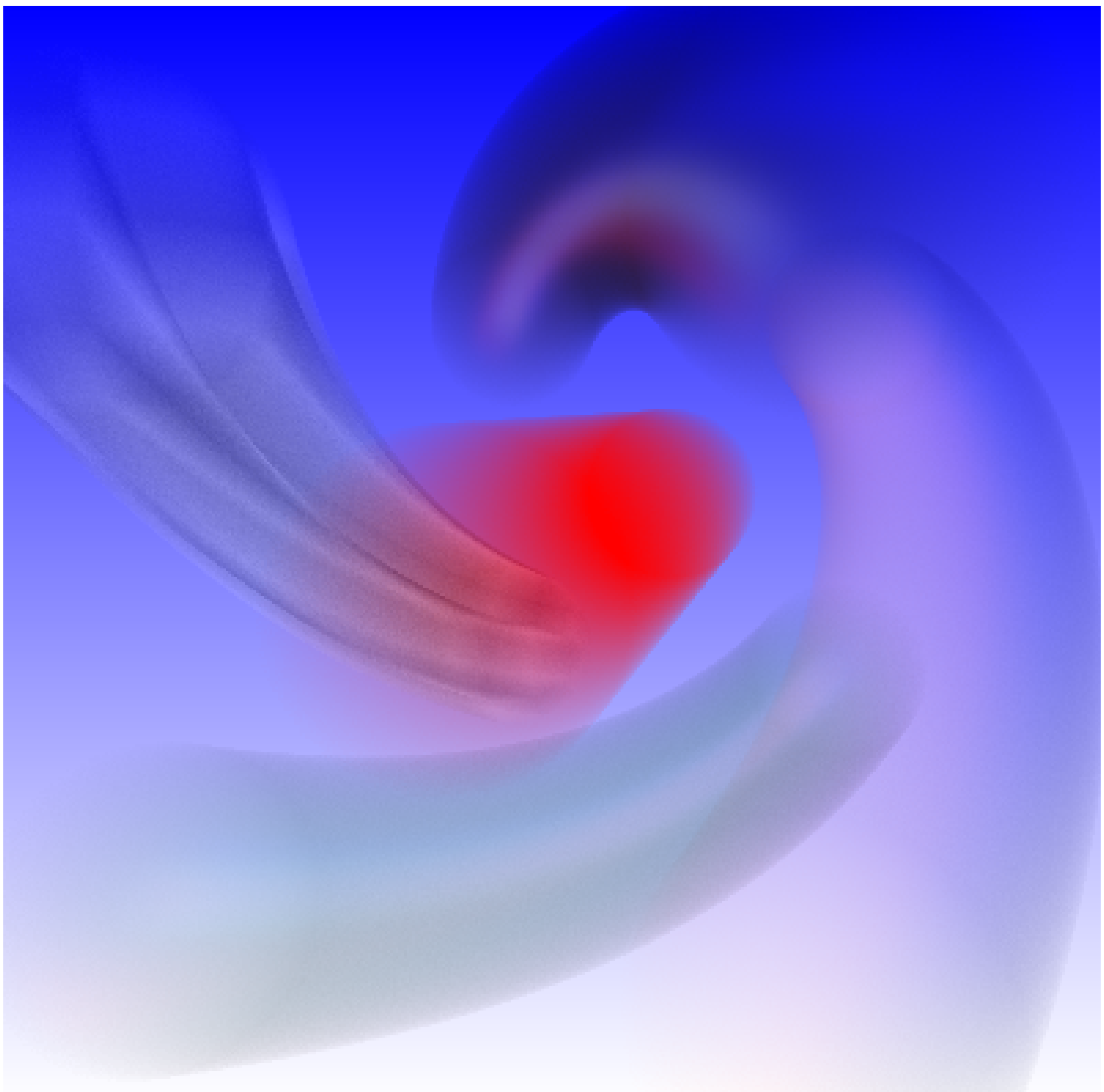


Figure 5.12: Combination Scene (Coupled SRS - box filter, 100 samples/pixel): 8:00 rendering time

Figure 5.13: Combination Scene (Decoupled MDF3/OBS - Gaussian filter): 1:48 rendering time

Chapter 6

Conclusions and Future Work

In this thesis, we have examined the motion blur problem as that of integrating an intensity function over time. We used point sampling as a general tool for computing the value of this integral. We separated the intensity function into two components, visibility and shading, and explored different strategies for sampling each. We now state our conclusions with respect to visibility and shading, then discuss future work.

6.1 Conclusions

6.1.1 Visibility

In Chapters 3 and 4 we explored the visibility function, which determines the object that is visible through a particular image plane position at a particular time. We saw that this function is discontinuous, and explored sampling strategies for approximating it. We saw that non-adaptive sampling strategies such as Stratified Random Sampling and Uniform Sampling produce banding artifacts and/or excessive noise when applied to the visibility function. We found that adaptive algorithms that actively search for the discontinuities produce a far better approximation of the visibility function for the same computational cost. Furthermore, we found that the visibility function exhibits a high degree of pixel-to-pixel coherence. Using principles of Ouellette and Fiume's Random Seed Bisection algorithm [11] and exploiting this coherence, our adaptive MDF3 algorithm efficiently produces a smooth approximation of the visibility function, resulting in images that are free of excessive noise or banding artifacts.

6.1.2 Shading

In Chapter 5, we examined the shading component of the motion blur integrand, that is, variations in image intensity over time as different parts of each object move through the pixel. We found that by using a good visibility solution (produced by MDF3) as a guide, the non-adaptive sampling strategies (Uniform Sampling, Stratified Random Sampling) that were unsuitable for visibility approximation can in fact produce good estimates of the shading integrals. By applying these strategies within each object's visible interval rather than over the exposure time interval as a whole, good estimates of the final pixel colour can be obtained. This "decoupled" approach also allows us to tune the shading quality parameters of each object individually, resulting in more efficient rendering, especially in scenes containing objects with widely varying shading characteristics.

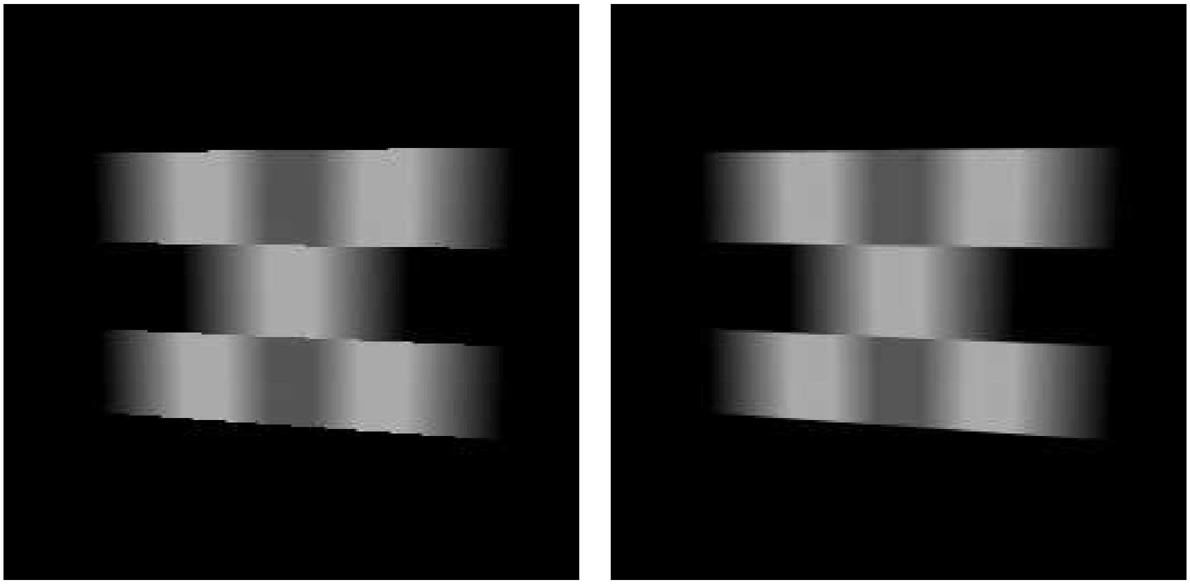
6.2 Future Work

This work is capable of producing high-quality motion-blurred images in a number of cases, but it lacks several features that would have to be added in order for it to displace Distribution Ray Tracing as the point-sampling approach of choice for high-quality CG production work. Four important effects not handled by our algorithms are: spatial filtering for moving objects, spatial filtering for stationary objects, transparency, and secondary effects. All of these effects can be simulated using Distribution Ray Tracing, but the sample counts required to do so are often quite high. It is not clear whether our algorithms can be extended to handle these effects.

6.2.1 Spatial Filtering for Moving Objects

We originally formulated the motion blur integral in Chapter 2 as a 3-dimensional integral of the image intensity function over the entire "space-time pixel". In Chapter 3, we decided to simplify this to a one-dimensional integral of image intensity seen through the pixel centre over time, thereby ignoring the spatial variation in intensity over each pixel. All of the temporal sampling techniques developed in this thesis have been targeted at solving this simplified one-dimensional integral.

In some cases, however, spatial intensity variations cannot be ignored, and this simplification is inappropriate. In particular, sharp details on an object (be they geometric edges or shading function discontinuities) that run parallel to the object's direction of motion will appear "jagged" in the final image. Figure 6.1 illustrates this using an example of a moving checkered square. In order to eliminate these jagged artifacts, we must filter (blur) the intensity function over space as well as time. As described in Chapter 2, Distribution



No spatial filtering **Spatial filtering**

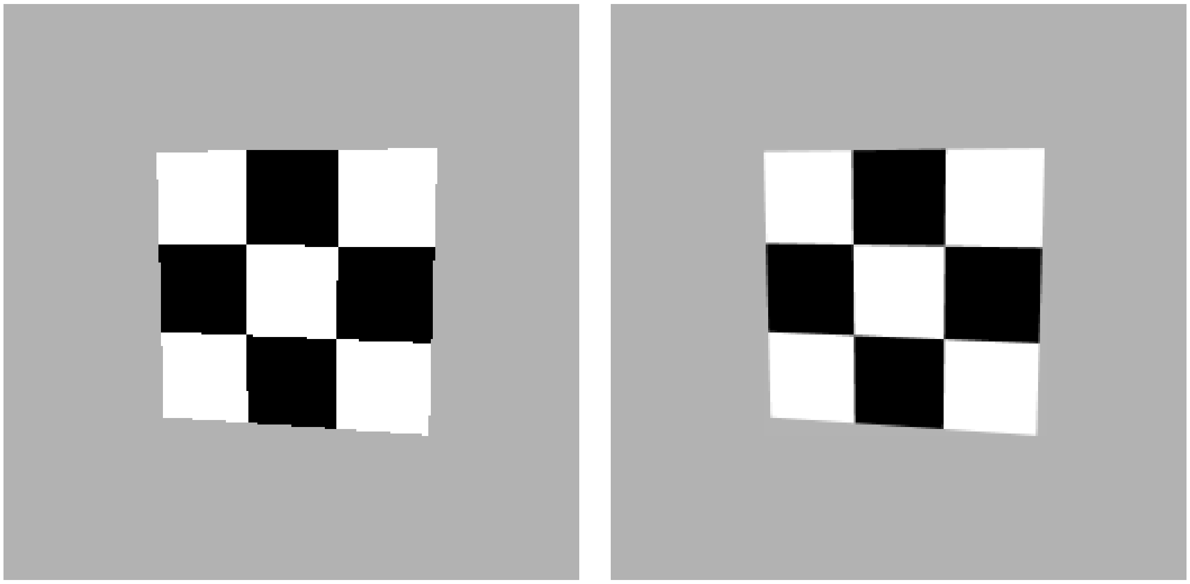
Figure 6.1: Example illustrating the need for spatial filtering of the intensity function for a moving object. The top and bottom edges of the object and the horizontal edges of the checker texture all run parallel to the object's direction of motion. Without spatial filtering, these edges appear jagged in the left image, which was produced using our algorithm. The right image (produced by Distribution Ray Tracing) shows that by incorporating spatial filtering, these artifacts are eliminated.

Ray Tracing applies Stratified Random Sampling to filter the full 3-dimensional intensity function over space and time. The second image in Figure 6.1 was produced using this algorithm, and it demonstrates that spatial filtering can eliminate these artifacts.

We are uncertain whether full 3-dimensional sampling of the intensity function is required to achieve this result for moving objects. We suspect that by distributing the spatial location of our samples along the direction perpendicular to the object's motion through the pixel, we could filter the troublesome details. This two-dimensional strategy amounts to sampling the intensity function along a plane through the space-time pixel. More work would need to be done to determine the merits and feasibility of this approach.

6.2.2 Spatial Filtering for Stationary Objects

Virtually all of this thesis has concerned itself with sampling the intensities of moving objects over time. With a stationary object, our samples in time are wasted, since the object's intensity does not change over time! Rather, stationary objects require samples to be distributed in space, in order to filter their stationary geometric edges and shading discontinuities. Figure 6.2 shows the result of applying our algorithm and Distribution Ray Tracing to a stationary checkered square. Because our algorithm considers only one spatial sample position per pixel, all edges in the object appear jagged. Distribution Ray Tracing, however,



No spatial filtering **Spatial filtering**

Figure 6.2: Example illustrating the need for spatial filtering of the intensity function for a stationary object. Without spatial filtering, all the edges of the object appear jagged. Spatial filtering eliminates these artifacts

distributes its samples over the entire pixel area, thereby eliminating these artifacts.

Spatial filtering of stationary objects could conceivably be incorporated into our algorithm. When a stationary object is known to be visible in a pixel, we would need to modify our visibility and shading sampling algorithms to distribute that object's samples in space rather than time. This would solve most problems related to stationary objects, but it would break down in some situations. Consider for example a spinning checkered sphere. The object is "stationary" in a sense because its silhouette does not change in the image, so we would need to distribute our visibility shading samples in space to smooth the edges of the sphere. On the other hand, the texture on the object is in motion, so it would require shading samples to be distributed in time in order to blur it. Detection and handling of hybrid cases such as this would be difficult.

6.2.3 Transparency

Throughout this thesis, we have assumed that the visibility function is single-valued, that is, that only one object is visible at any particular point in space and time. This is true if the scene is made up entirely of opaque objects, but the assumption breaks down when transparency is introduced. With multiple objects simultaneously visible through a pixel centre, we must track the visibility discontinuities resulting from each object. This could be accomplished by having the visibility function return a sequence of values representing the visible objects, ordered from front to back. By locating the points at which this visibility sequence changes, the visibility function for the entire pixel could be reconstructed. By modifying our shading algorithms to shade all visible objects, we could achieve good results.

6.2.4 Secondary Effects

In this thesis, we have only considered "primary" motion-blur effects, that is, blur resulting from observing objects directly through the camera lens. In fact, a moving object's blur can manifest itself in many other forms in the scene. For example, shadows cast by a moving object are motion-blurred, as are reflected and refracted images of the object. These effects are known as "secondary" motion blur effects, since they are in essence projections of the moving object's blur onto other objects in the scene. Point sampling can be used to simulate all of these effects in a raytracer by simply distributing secondary rays (shadow rays, reflected/refracted rays) in time. Just as in the primary case, improper distribution of these samples can lead to various artifacts in the secondary effects. For example, Distribution Ray Tracing uses Stratified Random Sampling to distribute secondary rays in time, and therefore suffers from excessive noise and banding artifacts in shadows/reflections/refractions of moving objects if too few samples are used.

The quality of these secondary effects could definitely be improved using our adaptive sampling algorithms, but the extra associated expense of doing so could be prohibitive. For example, to properly motion-blur shadows of a moving object from several light sources, we would have to apply MDF3 separately for each light source! Furthermore, we would probably have to maintain separate visibility caches for each object onto which motion-blurred shadows are cast, in order to properly exploit pixel-to-pixel coherence. More investigation would be needed to determine if the added expense and complexity of this approach are worthwhile, given that secondary effects are less noticeable in the final image than primary effects.

6.3 Summation

Thus, it seems clear that our algorithm will not displace Distribution Ray Tracing without significant work to address the wide variety of primary and secondary phenomena handled by this algorithm. If the spatial filtering issues with MDF can be solved, though, MDF could conceivably replace Stratified Random Sampling as the sampling strategy of choice for primary blur effects. It is likely that Stratified Random Sampling will remain the algorithm of choice for secondary effects, due to the aforementioned difficulties in applying MDF for these effects. It is the hope of the authors that a hybrid algorithm employing MDF for primary effects and SRS for secondary effects will be developed in the future.

Bibliography

[1] Edwin Catmull. An analytic visible surface algorithm for independent pixel processing. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 109–115, July 1984.

[2] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 279–288, August 1993.

[3] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. volume 21, pages 95–102, July 1987.

[4] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 137–45, July 1984.

[5] Eliot A. Feibush, Marc Levoy, and Robert L. Cook. Synthetic texturing using digital filters. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, pages 294–301, July 1980.

[6] Charles W. Grant. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 79–84, July 1985.

[7] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In Forest Baskett, editor, *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 309–318, August 1990.

[8] J. Korein and N. Badler. Temporal anti-aliasing in computer generated animation. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 377–388, July 1983.

[9] Nelson Max. Polygon-based post-process motion blur. *The Visual Computer*, 6(6):308–314, December 1990.

[10] Nelson L. Max and Douglas M. Lerner. A two-and-a-half-D motion-blur algorithm. In B. A. Barsky, editor, *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 85–93, July 1985.

[11] Marc J. Ouellette and Eugene Fiume. Approximating the location of integrand discontinuities for penumbral illumination with linear light sources. In *Graphics Interface*, pages 66–75, June 1999.

[12] M. Potmesil and I. Chakravarty. Modelling motion blur in computer-generated images. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17, pages 389–399, July 1983.

[13] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997.

[14] M. M. Wloka and R. C. Zeleznik. Interactive real-time motion blur. *The Visual Computer*, 12(6):283–295, 1996. ISSN 0178-2789.