Diploma Thesis

Motion Blur

Leonhard Grünschloß

April 3, 2008

Faculty of Engineering and Computer Sciences
Ulm University, Germany

First supervisor: Dr. Alexander Keller
mental images GmbH
Berlin, Germany

Second supervisor: Prof. Dr. Michael Weber
Institute for Media Computer Science
Ulm University, Germany

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von mir angegebenen Quellen angefertigt zu haben. Die Arbeit wurde noch keiner Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Leonhard Grünschloß (Matrikelnr. 517015), Ulm, 3. April 2008

Acknowledgements

I would like to express my sincere gratitude to Alexander Keller. He always encouraged me and provided invaluable feedback and suggestions. His ongoing confidence in me made this thesis possible.

Next I would like to thank the members of the computer graphics group at Ulm University: Holger and Sabrina Dammertz, Manuel Finckh, Johannes Hanika, and Sehera Nawaz. I will miss our discussions and movie-seminars! Many thanks also to Matthias Raab, Daniel Seibert, and Carsten Wächter at mental images for their help.

My parents have provided guidance and support thoughout my studies – thank you very much for everything!

Finally I would like to thank Claudia Nasenberg for all her love and care.

Contents

1 Introduction

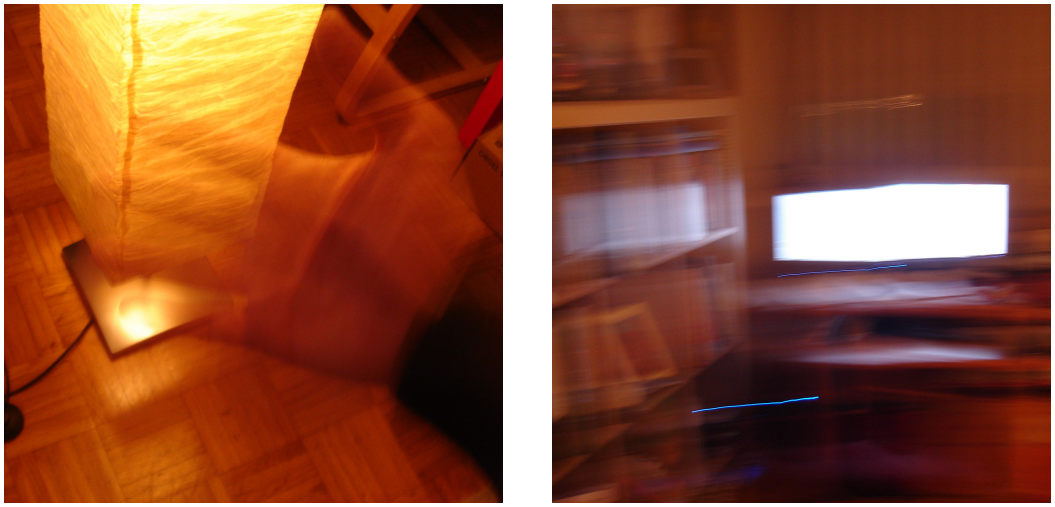
Although first approaches to simulate motion blur in realistic image synthesis have already been presented about three decades ago, it is still one of the hardest challenges for current production renderers. Before we can treat the algorithmic concepts behind motion blur simulation we start by describing how motion is perceived by the viewer.

1.1 Perception of Motion

Depending on the kind of image sensor, motion is perceived differently. Especially we need to distinguish between cameras (where the effect of motion blur is rather similar for both analogue and digital ones) and the human eye.

1.1.1 Cameras

Motion blur occurs in photography whenever either the objects or the camera moves during the shutter interval (see Figure 1.1). Motion that occurred over this extended period of time is visible in a single snapshot, resulting in a distinctive blurry streak for fast moving objects.



(a) Object movement. (b) Camera movement.

Figure 1.1: Different sources for motion blur in photographs.

Motion blur is an important visual clue for the occurrence of fast movements. Whenever rendered imagery does not simulate motion blur, a sequence of images appears to strobe.

Hence audiences will instantly recognize the imagery as "fake" [AGP+95]. Thus motion blur simulation is a crucial feature for every production renderer. [1]

Motion blur can be seen as the result of temporal anti-aliasing. Without it, strange effects such as wheels spinning backwards on a wagon occur due to the insufficient sampling rate. Perceptual temporal strobing on common display devices with about 24 frames per second is greatly reduced by integrating radiance over time. The resulting filtering effect of motion blur therefore is essential for fast-moving animated scenes.

1.1.2 Human Visual System (HVS)

Interestingly, a very similar effect can be observed in the human visual system (HVS). Due to the retinal information decay of photoreceptor response, the visual information is accumulated over an interval of approximately 125 ms [Zag97].

It is rather clear how a motion blurred image should ideally look if a simulation of a camera is desired. However such an image might differ extremely to what the human eye captures. Partly that is caused by the motion tracking abilities of the HVS. For example, if a ball moves predictably in front of a static background, the eye tracks the ball. Consequently, the background gets blurred while the ball remains sharp. A camera would capture the opposite effect instead [Zag97]. Animators therefore must often carefully fine-tune motion blur effects to make them "look right". For the remainder of this work we will concentrate on the easier task of simulating motion blur as a camera would capture the effect.

1.2 Motion Blur is a Mathematically Hard Problem

All production renderers take a significant performance hit when motion blur needs to be simulated correctly. That is mainly caused by the fact that we need to determine object visibility both in screen space and time (the spatio-temporal domain). Additionally, due to object and camera movement, the appearance of objects can vary drastically during the shutter interval. Most approximations fail for at least one of the following scenes, partially mentioned in [CPC84]:

- Visibility changes within a single object: A part of an rotating object occludes some other part during for some part of the shutter interval. This is difficult for post-processing approaches that rely on all moving parts being visible on a still image.
- A moving object is occluded by another moving object in front of it. The occluded object should not be visible, especially in the motion blurred regions of the object in front. This poses problems for algorithms that try to add motion blur for each object separately and composit the results afterwards.
- Non-linear movement functions, e.g. rotating propellers and high-order curves cannot approximated sufficiently using few linear segments. Motion blur methods relying on these approximations may get very expensive when many linear segments are used.

[1] Interestingly, the "inverse" problem also exists: Given a motion blurred camera image, compute an unblurred image, i.e. filter out the effect of unintentional movement while the shutter of the camera was open [YSQS07].

- Moving objects with non-zero genus (e.g. a torus) are be difficult for algorithms that try to find silhouette edges or sweep surfaces to substitute geometry to simulate motion blur by static objects.
- Object deformation, e.g. squashing a sphere during exposure. Consequently, the motion for every vertex might be different. Algorithms relying on single motion vectors for objects cannot handle such transformations.
- Highlights on rotating objects: The highlight should stay focused if the lights are static. The object texture should become blurry, though. Such a setting is problematic for algorithms that try to clamp the shading frequency. A similar scene consists of static caustics cast on a moving ground. Extreme shading changes can also be caused by moving objects, the camera and lights at the same time.
- Moving objects that intersect during the shutter interval are difficult since it complicates visibility determination and breaks compositing approaches.
- "Triangle soups" are difficult for algorithms that rely on a small number of distinct objects, respectively a mapping from triangles to few parent objects.
- Secondary effects, such as motion blurred shadows and reflections are generally difficult to handle, especially combined with global illumination.

2 Specifying Motion

Before the motion blur simulation takes place, we need to specify the movement of both the objects and the camera. Motion can either be recorded from real objects or modeled artificially.

2.1 Motion Capture (MoCap)

Motion capture is a technique to record movement of physical objects. It is famous for being used in the production of movies and video games, where the motion of real actors is digitally recorded. This data is then used to animate virtual characters. However motion blur can also be used to track the motion of everything else: animals, trees in the wind, facial expressions, etc.

The details on how the motion is recorded differs for each mocap system. Usually, numerous markers are attached to the object that is to be recorded. Multiple cameras then record the performance from different angles. Computer vision algorithms are used to find corresponding markers in the recorded videos and the three-dimensional coordinates of each marker can then be computed by triangulation.

The advantages of such an approach are obvious: Real motion can be transferred directly to computer generated objects (typically even in real-time) instead of manually animating objects, which is a very tedious and time-consuming work.

Unfortunately, high quality mocap equipment is still very expensive. There are also many situations when mocap simply is not practical since the markers cannot be attached, the scene is impropriate to be captured using static cameras or suitable actors are not available.

2.2 Simulation

Motion capture is not suited effects in artifical environments, e.g. the motion of water, realistic cloth movement or falling rocks that respond to virtual forces. Similarly, motion capture is not practical for crowd movement simulation. Yet modeling these effects manually is very time consuming and realistic results are difficult to achieve.

Given the right boundary conditions, *physics-based animation* tries to solve partial differential equations (PDEs) that formalize physical laws governing the world around us. More details on computer animation principles can be found in [SAG+05, Ch. 16].

2.3 Modeling Motion

A popular approach of animators is to use *keyframing* to control aspects of the scene. The animator defines the position, orientation, shape and shading of each object at a given set of moments. The computer then interpolates between those settings, creating in-between frames.

Object transformations are often given as real-valued 4×4 matrices containing both translation and rotation. While such a matrix representation is useful for composing different transformations by matrix multiplication, it is difficult to interpolate between two matrices since interpolating the components independently generally does not produce reasonable results.

Instead, the matrices usually are decomposed into a translation, scaling and rotation component [SD92]. Source code for such a matrix decomposition can be found in [Sho94]. All components can then be interpolated separately for intermediate times.

Alternatively, [Ale02] has described a method for interpolating transformation matrices directly without decomposition by defining meaningful scalar multiples and commutative addition of transformation matrices. Corrections to the paper describing this approach can be found in [BBM04].

In Subsection 2.3.1 we describe how positions of objects can be described using trajectory curves for translation. Their orientation is typically given by rotations, as outlined in Subsection 2.3.2.

2.3.1 Position by Translation

Complex trajectory paths are often exported from modeling programs as *non-uniform rational B-spline (NURBS) curves* because of the great variety of shapes they are able to represent.

Definition 1 ([PT97, p. 50])**.** Let the *knot vector* $\mathbf{U} = (u_0, \ldots, u_m)$ be a nondecreasing sequence of real numbers with $u_i \leq u_{i+1}$, $i = 0, \ldots, m-1$. The i-th *B-spline basis function* of p-degree is defined as

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u < u_{i+1}, \\ 0 & \text{otherwise,} \end{cases} \tag{2.1}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u). \tag{2.2}$$

Note that subsequent knots may be equal and Equation 2.2 may yield the quotient $\frac{0}{0}$. This quotient is defined to be zero.

An efficient implementation of the B-spline basis functions uses two observations:

- Since the support of the rational basis functions $R_{i,p}(u)$ is finite (see Figure 2.1), the first step of evaluating a NURBS curve is to determine the knot span in which u lies. As the values in $\mathbf{U}$ are nondecreasing, a binary search algorithm can be used.
- The recursive definition in Equation 2.2 can be rewritten as an inverted triangular scheme. This leads to the following iterative algorithm with the additional benefit of

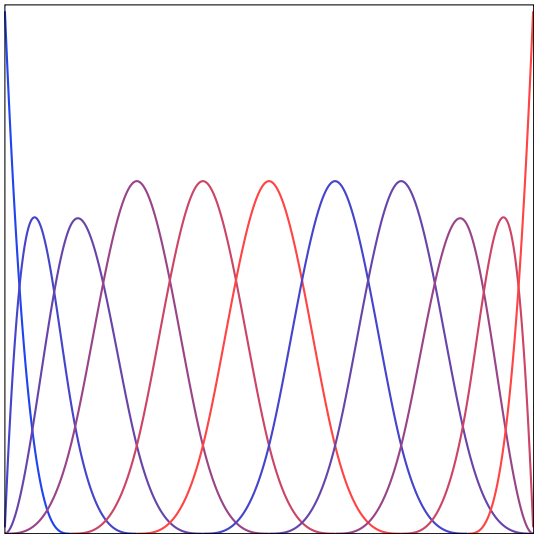


Figure 2.1: Cubic B-spline basis functions for the knot vector $(0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 8, 8, 8)$.

avoiding the problem of dividing by zero [PT97, p. 68]. For the curve parameter $u \in [u_i, u_{i+1})$, it computes all p nonvanishing B-spline basis functions.

```
inline void basisFunctions(const unsigned int i, const float u) {
  bfunc[0] = 1.f;

  for (unsigned int j = 1; j <= degree; j++) {
    left[j] = u - knots[i + 1 - j];
    right[j] = knots[i + j] - u;
    float saved = 0.f;

    for (unsigned int r = 0; r < j; r++) {
      const float tmp = bfunc[r] / (right[r + 1] + left[j - r]);
      bfunc[r] = saved + right[r + 1] * tmp;
      saved = left[j - r] * tmp;
    }

    bfunc[j] = saved;
  }
}
```

Definition 2 ([PT97, p. 117f]). A p-th degree *NURBS curve* is defined by

$$\mathbf{C}(u) = \sum_{i=0}^{n} R_{i,p}(u)\mathbf{P}_i := \frac{\sum_{i=0}^{n} N_{i,p}(u) w_i \mathbf{P}_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i}, \quad a \le u < b, \tag{2.3}$$

where $\mathbf{P}_i$ denotes the *control points*, $w_i \ge 0$ are the *weights* and $N_{i,p}(u)$ are the p-th degree

B-spline basis funtions defined on the knot vector

$$\mathbf{U} = (\underbrace{a, \ldots, a}_{p+1 \text{ times}}, u_{p+1}, \ldots, u_{m-p-1}, \underbrace{b, \ldots, b}_{p+1 \text{ times}}). \tag{2.4}$$

Important properties:

- The repetition of the knots at the beginning and end of the knot vector ensures that the curve begins at the first and ends at the last control point.
- Setting a weight w_i to zero effectively disables the control point $\mathbf{P}_i$.
- Nonnegativity: $R_{i,p}(u) \geq 0$ for all i, p and $u \in [a, b)$.
- Partition of unity: $\sum_{i=0}^{n} R_{i,p}(u) = 1$ for all $u \in [a, b)$.
- Local support: $R_{i,p}(u) = 0$ for $u \notin [u_i, u_{i+p+1})$.
- Strong convex hull property: $\mathbf{C}(u)$ lies within the convex hull of the control points $\mathbf{P}_{i-p}, \ldots, \mathbf{P}_i$ for $u \in [u_i, u_{i+1})$.

NURBS curve allow to specify the shape of the animation curve of moving objects in a very flexible way. However, this does generally not include control over how fast an object is moving along this curve. For example, if an object is supposed to move along the curve with constant speed, it is not sufficient to interpolate the curve parameter u linearly with time (see Figure 2.2).

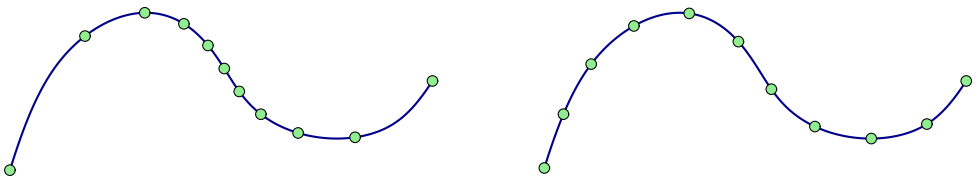


Figure 2.2: Different parametrizations of a NURBS curve. On the left, the circles denote points $\mathbf{C}(u)$, where u is equidistantly spaced in the parameter range. On the right, the circles denote points $\mathbf{C}(u(s(t)))$, where $s(t)$ maps linearly from time to arc length. $u(s)$ returns the curve parameter for a given arc length s.

A convenient way to control the speed of an object along the curve is to specify a function $s(t)$ that maps the elapsed time to arc length along the curve. Given a function $u(s)$ that maps arc length to the corresponding curve parameter, this allows for a very flexible setup. Unfortunately, such a reparametrization to arc length is nontrivial. In general, no planar curve, other than a straight line, can be parametrized by rational functions of its arc length [FS91].

This means we have to resort to numerical methods. A discussion of efficient reparametrization with error bounds can be found in [Blo99]. An algorithm that uses adaptive Gaussian

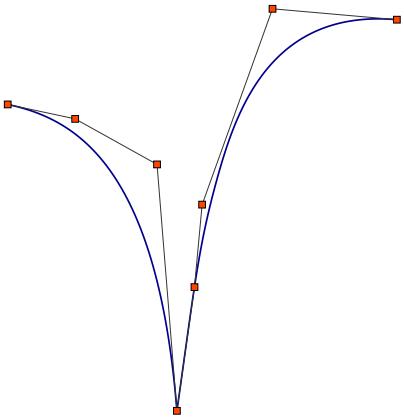


Figure 2.3: A knot multiplicity equal to the degree of the NURBS curve causes the curve to interpolate the control point at the bottom, where the curve is only C^0-continuous. The knot vector for this cubic B-spline curve is $(0, 0, 0, 0, 1, 2, 2, 2, 6, 6, 6, 6)$.

integration and works for general parametric curves whenever the derivative is available is described in [GP90b]. However, NURBS curves allow for discontinuous derivatives (see Figure 2.3) which might cause numerical problems.

The following algorithm is straightforward to implement and works well in practice for all C^0-continuous curves. We approximate the curve as a set of linear segments between points $\mathbf{C}(t_i)$ and $\mathbf{C}(t_{i+1})$ for sufficiently densely equidistantly spaced parameter values $t_i \in [u_1, u_m)$ with $t_i < t_j$ for all $i < j$ (see Figure 2.4).

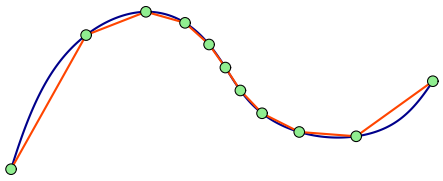


Figure 2.4: The arc length of a curve can be approximated using the sum of lengths of linear segments.

We then create a table of approximate arc lengths

$$\tilde{s}(t_i) = \begin{cases} 0 & \text{if } i = 0, \\ \tilde{s}(t_{i-1}) + \|\mathbf{C}(t_i) - \mathbf{C}(t_{i-1})\| & \text{otherwise.} \end{cases} \tag{2.5}$$

To quickly find the corresponding curve parameter u for a given arc length s, we exploit the fact that the values of the table above are nondecreasing. Thus, a binary search is feasible to find the parameter values t_i, t_{i+1} with $\tilde{s}(t_i) \leq s < \tilde{s}(t_{i+1})$. To improve the precision of this

approach, a linear interpolation can be used:

$$u(s) \approx t_i + \frac{u - u_i}{u_{i+1} - u_i}(t_{i+1} - t_i). \tag{2.6}$$

Since the reparametrization by arc length will potentially be evaluated millions of times for the synthesis of a single image, the binary search might turn out to be a major bottleneck. However, using the above arc length table, we can approximate the curve using linear segments *of the same length* in a preprocessing step (see Figure 2.5). For best accuracy, the number of linear segments used to build the table in Equation 2.5 should exceed the number of these linear segments of equal length. By storing the endpoints of these segments, we also do not need to apply the algorithm to evaluate Equation 2.2 anymore.

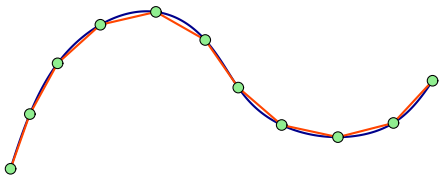


Figure 2.5: The curve can be approximated by linear segments of the same length to allow for fast lookup by arc length.

Any query for a point on the curve parametrized by arc length can now be answered in constant time: Given the table of linear segments, each of length p, the index of the linear segment for arc length s is given by $\left\lfloor \frac{s}{p} \right\rfloor$. The corresponding point on this linear segment can again be found by linear interpolation. We would like to stress that while the endpoints of these linear segments lie exactly on the curve, the points in between may not.

This approximation also leads to potentially tighter spatial bounds. While we know that the exact curve lies within the convex hull of all control points, we know that the approximation lies within the convex hull of all endpoints of the linear segments. The latter volume might be significantly smaller.

2.3.2 Orientation by Rotation

Complementing NURBS curves, quaternions can be used to specify object orientation [Sho85]. Quaternions consist of four values that describe rotation: three give the coordinates for the axis of rotation while the fourth determines the angle of rotation.

It is not difficult to convert rotation matrices from and to quaternions, which also form a non-commutative group under multiplication. Yet in contrast to rotation matrices, quaternions can easily be interpolated (if constant velocity and minimal torque is desired, this is called a spherical linear interpolation, also known as “slerp”). Additionally, they avoid the gimbal lock problem that can occur when Euler angles are used to specify orientations.

To interpolate more than two quaternions the concept of spline curves can be extended to quaternions to generate smooth paths through rotation space [BCGH92, KKS95].

Bounding Boxes for Rotations

We sometimes need to find tight enclosing axis-aligned bounding box (AABB) for an object under continuous rotation. For example, a bounding volume hierachy built over the moving objects needs such information. This problem is equivalent to finding a tight AABB for a set of circular arcs.

Any 3-dimensional rotation about an arbitrary axis **a** by angle θ can be represented by the composition of three basic operations:

1. Rotate **a** to align with the the canonical axis **z**,
2. rotate by θ about **z**,
3. rotate **z** back to **a**.

Thus, without loss of generality, we may restrict ourselves to the case of rotating about **z**. If we know the enclosing AABB for the rotation about **z**, we simply need to rotate all endpoints back to the previous coordinate system and find the extremal coordinates to get back an axis-aligned representation. When rotating about **z**, we know that a point (u, v, w) gets transformed to $(\bar{u}, \bar{v}, w)$, where

$$\bar{u} = u\cos\theta - v\sin\theta, \tag{2.7}$$
$$\bar{v} = -u\sin\theta - v\cos\theta. \tag{2.8}$$

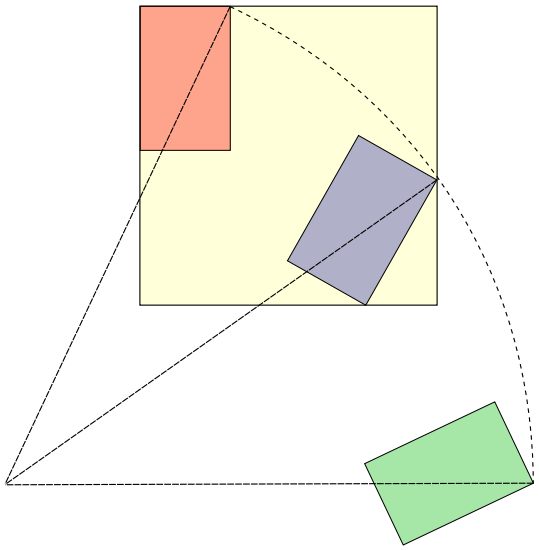


Figure 2.6: For a rotation of the red rectangle, we consider each corner point separately. The maximum coordinate along the **x**-axis for the upper-left corner point is obtained at the green rectangle. However, if the rotation stops before, the maximum is obtained for the rotational endpoint at the blue rectangle. The yellow rectangle denotes the resulting AABB for the continuous rotation.

To find values of θ where $\bar{u}$ and $\bar{v}$ are maximal (see Figure 2.6), we differentiate both of the above equations with respect to θ:

$$\frac{d}{d\theta}\bar{u} = -u\sin\theta - v\cos\theta, \tag{2.9}$$

$$\frac{d}{d\theta}\bar{v} = u\sin\theta + v\cos\theta. \tag{2.10}$$

Setting both derivatives to zero reveals the angles for the maxima:

$$-u\sin\theta - v\cos\theta = 0 \tag{2.11}$$

$$\Rightarrow \frac{\sin\theta}{\cos\theta} = -\frac{v}{u} \tag{2.12}$$

$$\Rightarrow \theta = \arctan -\frac{v}{u}, \tag{2.13}$$

$$u\sin\theta + v\cos\theta = 0 \tag{2.14}$$

$$\Rightarrow \frac{\sin\theta}{\cos\theta} = \frac{u}{v} \tag{2.15}$$

$$\Rightarrow \theta = \arctan\frac{u}{v}, \tag{2.16}$$

where we required $\cos\theta \neq 0$. For $\cos\theta = 0$, the maximum can only be achieved if $u = 0$, respectively $v = 0$. Interestingly, this special case is implicitly handled in the implementation, since the arctan function of the C library maps $\pm\infty$ arguments to $\pm\frac{\pi}{2}$.

If the continuous rotation ends before arriving at the values of θ derived above, the maximal coordinates are reached at the end of the rotation. This results from rotation being a continuous transformation. The minima are obtained by adding π to the angles corresponding to the maxima.

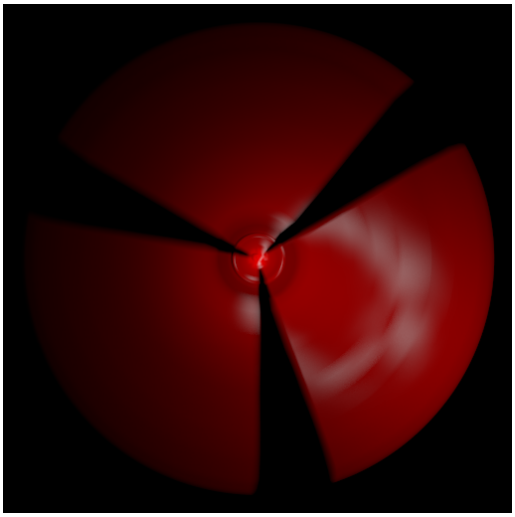
2.4 Practical Approximation by Linear Segments

In practice, motion trajectories are often approximated using a small number of linear segments. This approximation is motivated by the Taylor expansion and results in linear trajectories of object vertices. Both mental images' mental ray and Pixar's PhotoRealistic RenderMan currently handle motion this way. There are a number of advantages using linear segments:

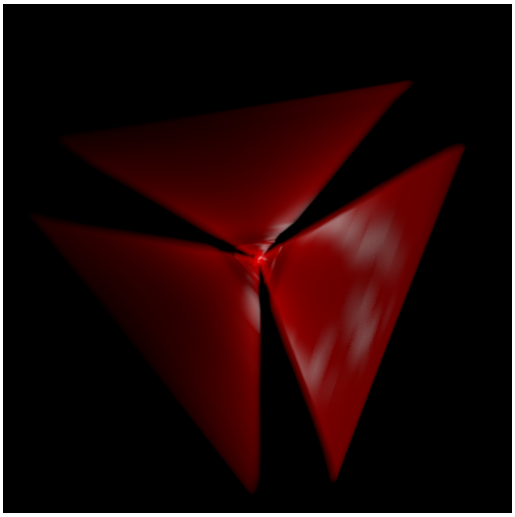
- Every kind of input curve can be approximated as accurately as desired using enough linear segments, so it is a very general approach.
- Linear interpolation is computationally cheap compared to evaluating high-order curves (and in addition much more numerically stable).
- The extrema of a multiple linear segments are easy to compute. For many high-order curves tight bounds are hard to achieve. This is important whenever bounding boxes of moving objects are needed (e.g. for building a spatial hierarchy).
- Linear segments exploit the inertia of objects [KK07] in the sense that constant velocity is assumed and consequently the error is proportional to the acceleration.

For motion blur, such an approximation often works well in practice. That is because during the rather short time interval of a single frame the trajectory of objects usually is very well approximated using a small number of linear segments. However non-linear trajectories, e.g. circular segments, polygons with a degree larger than one, and discontinuities are difficult to handle:

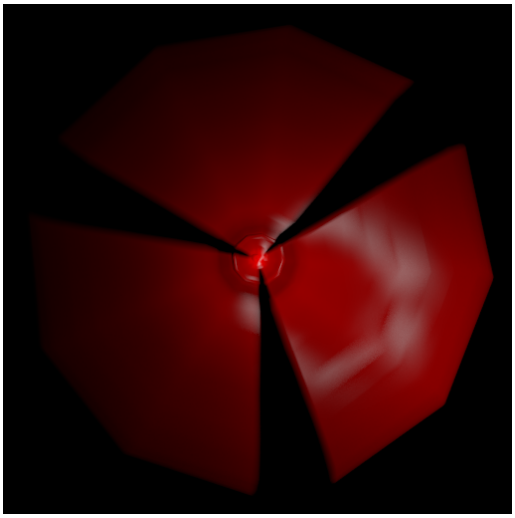
- Fast rotational movement requires a large number of linear segments for a sufficient approximation (see Figure 2.7).
- Discontinuities in the original trajectory curve need special care when the curve is approximated by linear segments. Imagine a ball jumping on the floor – it is important that endpoints of the linear segments are located at the positions where the direction of the ball changes instantaneously. Otherwise the ball appears to be floating above the ground (see Figure 2.8). For NURBS curves such discontinuity points can be found by analyzing the knot vector and the control points for multiplicity (see Figure 2.3).



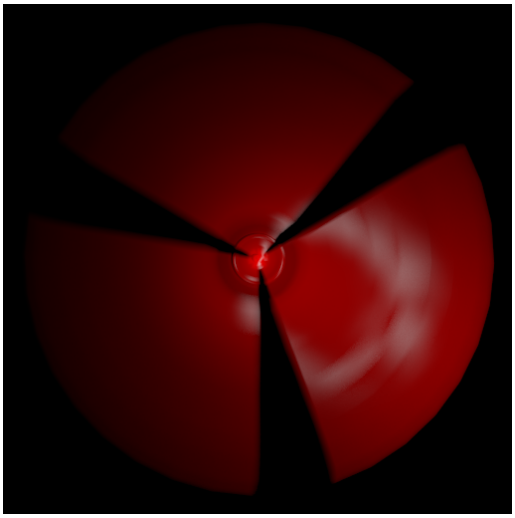
(a) The actual rotation.



(b) One linear segment.

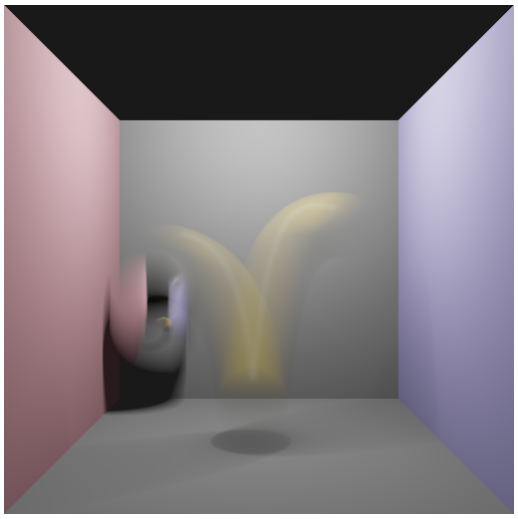


(c) Three linear segments.

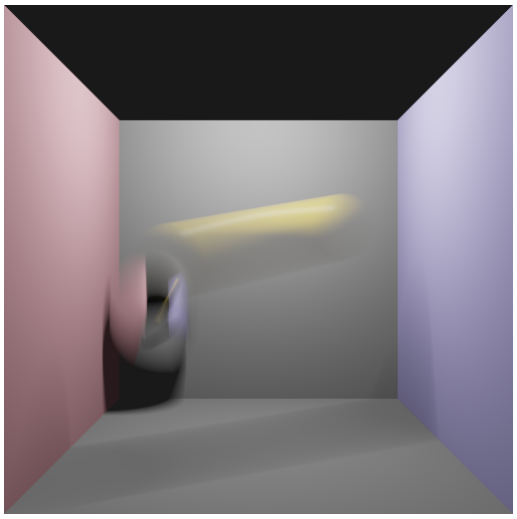


(d) Ten linear segments.

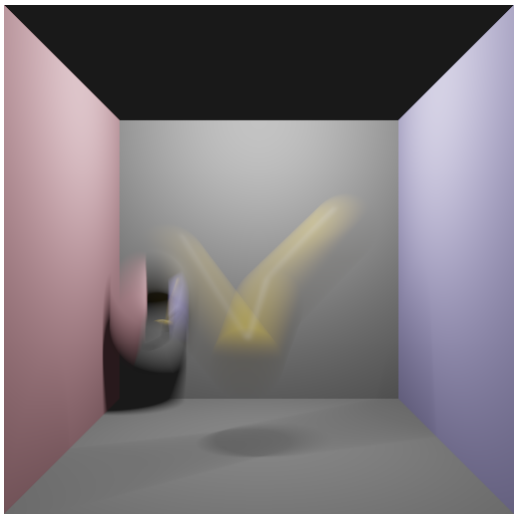
Figure 2.7: A large number of linear segments is needed to sufficiently approximate a fast rotation.



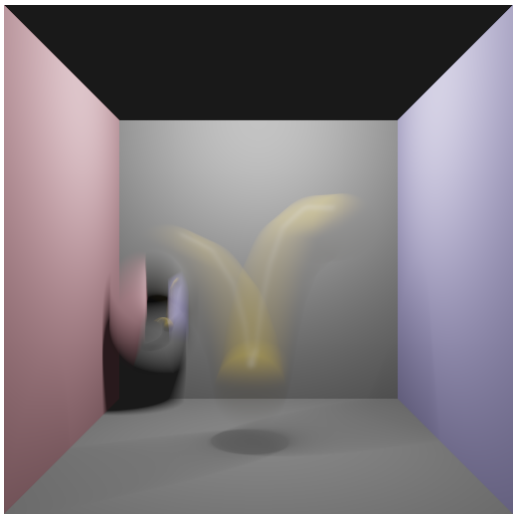
(a) The actual curve trajectory.



(b) One linear segment.



(c) Three linear segments.



(d) Ten linear segments.

Figure 2.8: If endpoints of the linear segments do not coincide with discontinuities of the trajectory curve, the ball appears to be jumping above the floor.

3 Rendering Motion Blur

With real cameras, the shutter takes some time to open and close [Ste05] and there are different kinds of aperture closings [Gla99]. While for some scenes it may be important to simulate these camera-dependent effects, for this thesis we assume ideal apertures that open and close instantly. Furthermore, we use a temporal box-filter, although higher degrees may yield superior results [IK00].

We assume the reader to be familiar with basic rendering concepts such as rasterization and ray tracing (see [SAG+05] for a nice introduction). [SPW02] contains an excellent survey and categorizes the problem of simulating motion blur into different methods. We follow this approach and extend it with newer publications.

Let $L(\mathbf{x}, t)$ denote the radiance arriving at the position $\mathbf{x}$ at time t on the image plane. If A denotes the area of a single pixel and T denotes the shutter interval, the color of a pixel (using a simple box-filter for both the spatial and the temporal domain) is given by the three-dimensional integral

$$I := \int_A \int_T L(\mathbf{x}, t) dt d\mathbf{x}. \tag{3.1}$$

For the sake of simplicity we left out the integration over the aperture (for depth of field) and volume (for participating media effects) in the integral above.

3.1 Numerical Methods Based on Sampling

The seminal paper on distributed ray tracing [CPC84] introduced the most general technique for simulating motion blur, using Monte Carlo sampling [Sob94]. N random discrete samples $\mathbf{x}_i \in_R A$ and $t_i \in_R T$ for $0 \leq i < N$ are used to estimate the integral of Equation 3.1:

$$I \approx \frac{|A||T|}{N} \sum_{i=0}^{N-1} L(\mathbf{x}_i, t_i). \tag{3.2}$$

Depending on the variance of the function L, the number of samples that need to be taken might be very large. Variance reduction techniques such as importance sampling are often used to increase the efficiency of the estimator.

One appeal of the Monte Carlo method is its generality and its applicability for high dimensional integrals. We only need to be able to evaluate L at discrete points in space and time. Acceleration structures used for for ray tracing need to be built in a way that allows such a point query to be evaluated efficiently (see Chapter 4).

3.1.1 Quasi-Monte Carlo Samples

Instead of using random samples it is often advantageous to use well-stratified deterministic samples (so called quasi-Monte Carlo samples) to increase the rate of convergence. New constructions for such points are covered in Chapter 5.

3.1.2 Method of Dependent Tests

Variants of the Monte Carlo method include the method of dependent tests. Instead of using independent samples t_i for each pixel, one can average multiple images, where for each image every pixel corresponds to the same t_i [KB83, HA90] (see Subsection 3.1.3).

Due to the correlation of neighboring pixels a large number of samples in time is needed for artifacts to disappear. This effect can be reduced drastically by using interleaved sampling [KH01], where neighboring pixels are given different t_i. However the distribution of these values of t_i can be chosen as a grid pattern pattern, so in practice multiple small-resolution images (or tiles) can efficiently be rendered in parallel and interleaved afterwards.

If a lot of samples share the same value t_i, it might be advantageous to build an acceleration structure for each such time instant. Rays corresponding to this time instant can then be traversed very efficiently. This process is repeated for the remaining time instants.

3.1.3 Hardware-Based Methods

The accumulation buffer [HA90] is present in all current graphics cards and can be used to simulate motion blur as well. A number of discrete images are averaged, each rendered with the camera and objects moved appropriately for the corresponding point in time. Haeberli and Akeley furthermore describe a clever approach for generating images when a high update rate is desired. Instead of rendering N images that get averaged for every frame, the frame that was drawn $N-1$ frames ago is redrawn, subtracted from the accumulation buffer and finally the next frame is drawn and added to the accumulation buffer. This yields the new final frame (using only 2 frame computations instead of N). If storing N frames in memory is possible, a ring buffer could be used instead. Only a single new frame needs to be computed then to yield another final frame, since the previous N frame are still available.

If N is rather small, strobing artifacts may arise due to the correlation of time samples on adjacent pixels. This can be alleviated by the use of interleaved sampling (see Subsection 3.1.2).

As discussed in [WGER05], motion blur is simulated using the accumulation buffer in nVidia's Gelato renderer. They also tried to render stochastically with a shader program, but that was very inefficient using current GPUs.

New hardware mechanisms implemented on top of of today's graphics hardware pipeline to render time-continuous triangles were proposed by [AMMH07]. By a time-continous triangle they mean the set of all triangles defined by a starting and end triangle and all linear interpolations in between. They propose a mechanism to efficiently rasterize such sets, using stochastic samples in time. This even allows for depth of field effects (see Subsection 3.3.5) and glossy reflection rendering by averaging multiple passes.

3.1.4 Adaptive Sampling

It seems reasonable to assume that the variance of $L(\mathbf{x}, t)$ is larger in regions of the image plane where motion blur is apparent compared to regions where only static objects are visible. The concept of adaptive sampling therefore tries to improve estimations by placing more samples in regions where variance is high. Therefore efficiency can often be increased by determining image regions where moving objects are visible. To accomplish this, a simple extension to a ray tracer consists of sending a single ray that has no time sample assigned in order to check if it passes the path of a moving object (e.g. by traversing a hierarchy of bounding boxes that enclose the moving objects). Super-sampling of the temporal domain is then only enabled according to the result of this test. Thus we use more rays where the integration domain is larger (due to the time dimension).

[NY03] points out a problem with this approach that gets increasingly noticeable the faster objects are moving: Only a certain percentage of rays hits the moving object – the rest is used to estimate the radiance coming from the static object behind the moving object. If the percentage of rays that hit the static object is large (as in the case of fast moving objects) we consequently get a much better estimation of the radiance coming from the static object compared to regions where we do not use temporal super-sampling (and hence use a smaller number of rays). Consequently the noise level in regions with motion blur is much lower compared to regions where temporal super-sampling was not used.

To avoid these visual inhomogeneities Neilson et al. propose a simple modification: Rays that miss the moving objects are *not used* to shade the static object behind it. Instead another estimator is used to approximate the radiance coming from the static object using the same number of samples as for regions with no visible motion blur. The results of these two estimators are then summed with weights according to the fraction of rays that hit the static object.

The analytic approach described in Section 3.2 should allow for more efficient means to compute the fraction of time the static object is visible (only a single – yet more expensive – traversal of the acceleration structure would be necessary).

3.1.5 Photon Mapping

The photon map [Jen01] is a popular choice for approximating global illumination as it usually yields noise-free images much earlier than path tracers. It has been extended by [CJ02] to account for motion blur. The basic idea is very similar to distribution ray tracing [CPC84]: In the first pass, for each photon, a uniformly distributed random time is picked and used for the whole photon path. The key point however is to store the time of each photon as an extra dimension in the photon map.

In the second pass, eye rays with uniformly distributed times are traced. To generate an estimate of the density of the radiant energy at the hit point, the nearest photons need to be found. However the nearest neighboring photons are now to be understood as both spatially as temporally near (a compact neighborhood of four-dimensional space-time). Typically this nearest neighbor search is implemented using a 4D kd-tree. Similar to the non-time-dependent photon map, smoothing kernels for both space and time can yield superior results. Ignoring

numerical problems such a time-dependent photon map can be considered a consistent estimator: As the number of photons increases, both the area and time spanned by the nearest neighboring photons approach zero.

The ideas from Subsection 3.1.4 can be transferred to the photon map [Nei03, Sec. 3.3]. Instead of assigning a random time to *every* photon, one first checks whether the photon passes through the path of a moving object. If this is not the case, the photon is stored without a time sample at the static object. Otherwise a larger number of temporally distributed photons is cast. In the second pass time-dependent photons as well as photons that are valid for the whole shutter interval need to be considered.

Interval Photons

Neilson observes that for static parts of the scene, the temporal stratification is not only unnecessary, but leads to an increased memory footprint. That is because more photons need to be shot to attain the same photon density for arbitrary points in time compared to completely static scenes. Therefore Neilson develops an approximation to determine valid time *intervals* for photons instead of singular points in time (resulting in *interval photons*). It is an approximation since the results of rays traced for a finite number of time samples are used to estimate visibility intervals.

Again we propose the analytic approach in Section 3.2, yielding an accurate (and possibly faster) method. In the first pass, photons with an interval corresponding to the full shutter time are sent from all static light sources. Each such photon is intersected with the geometry using the analytic visibility algorithm. The resulting visibility intervals are used to store the photons at the hit points. In the next photon scattering step, we need to consider two cases:

- The scattering surface is static. The photon with its corresponding time interval can then be scattered, keeping its interval.
- The scattering surface is dynamic, i.e. it is moving or changing its material properties during the photon time interval. We then need to resort to scattering singular time value photons. That is because the energy of the photon and its scattering direction might change during the interval. The same is true for initial photons on a dynamic light source.

The second pass of the photon map needs to consider both types of photons: Those with a corresponding time interval and photons with singular time values (i.e. coincident interval interval bounds). Building a good kd-tree over interval photons is likely to be more difficult than for photons with singular points in time, though.

3.1.6 Multidimensional Lightcuts

The basic idea of lightcuts [WFA+05] is to solve the rendering equation [Kaj86] by discretizing the integral into a sum over many point lights. This sum is approximated with a strongly sublinear cost by clustering the point lights and using representatives for each cluster instead.

Multidimensional lightcuts [WABG06] are an extension of [WFA+05] to incorporate effects such as motion blur, participating media, depth of field and spatial anti-aliasing. The

value of a pixel then equals an integral over multiple domains. The multidimensional lightcuts approach discretizes this integral into two point sets, namely gather points $\mathbb{G}$ (originating from the camera) and light points $\mathbb{L}$ (generated from the light sources), similar to bidirectional path tracers [Vea97].

Since it would be too expensive to approximate the integral by summing up all pairwise interaction between points in the set (with $|\mathbb{G}|\,|\mathbb{L}|$ computations), hierarchies over both sets are built: the gather tree and the light tree, respectively. Each node of these trees represents a cluster of all the gather or light points beneath them. The clusters are built using a perceptual metric and conservative error bounds in estimating the contribution of a cluster. Thus, this method is similar to approximations that have been used by physicists to simulate the n-body problem of particles interacting with each other.

To adaptively find clusters suited for approximation a product graph of the gather tree and the light tree is built. This is an implicit hierarchy over the set of all gather-light pairs. Each product graph node corresponds to the pairing of a node from the gather tree and another node from the light tree. It represents all pairwise interactions between points in their respective clusters. The product graph is not a tree itself. Two nodes in the product graph have a parent-child connection if they correspond to the same node in one tree and to parent-child nodes in the other tree. Leaf nodes of the product graph correspond to individual gather-light pairs.

In order to incorporate effects such as depth of field, participating media, spatial antialiasing and of course motion blur, the key insight is to consider the pixel as a whole [DBBS06]. That is, for each pixel the set $\mathbb{G}$ of gather points is distributed in volume, aperture, pixel area and time, by shooting multiple rays per pixel.

It is important to note that the shutter interval T is discretized into a fixed set of time instants for each frame (similar to the method of dependent tests). The strengths and intensities of gather and light points are then time vectors corresponding to these time instants. If lights or gather points do not exist for certain time instants, the corresponding vector components are set to zero.

For every pixel a so called cut through the product graph is iteratively refined with the goal to let the cut approximate the current pixel accurately. A cut through the product graph is defined as a set of nodes such that, for any leaf node, the set of all paths from the root to that leaf node will contain exactly one node from the cut. Such a cut is a valid partitioning of the gather-light pairs into clusters. Cluster contributions are in turn approximated by the selection of a representative gather-light pair.

Representative pairs are picked such that both gather and light points exist at the same time instant. When a node of the cut through the product graph needs to be refined, a random time instant is picked for one child, while the other reuses the parent's time instant. Thus the cut consists of gather-light pair representatives that are distributed over the fixed set of time instances.

The building of the gather and light trees, selection of the representatives and the errors bounds used are quite involved. We refer the reader to [DBBS06, WABG06] for more details. Caustics currently cannot be handled by the algorithm.

3.1.7 Frameless Rendering

Frameless rendering approaches [BFMZ94, Zag97] can be seen as temporal supersamplers that undersample the spatial domain. Therefore they also fit into the category of Monte Carlo integration approaches. For interactive applications, frameless rendering enables rendering of inexpensive low-quality motion blur. $(0, m, 2)$-nets and especially the Sobol' sequence (covered in Chapter 5) are very well suited for determining the pixels to update next as they are suited to generate seemingly random pixel permutations on-the-fly.

3.1.8 Eulerian Framework

There are two basic approaches when the movement of fluids or gases, such as water and smoke, are simulated. One can either let objects (typically small particles) carry quantities, such as position, acceleration and velocity. This is called the Lagrangian framework. In the Eulerian framework however the simulation domain is discretized into grids and the simulator observes the quantities at the fixed grid positions. Both approaches have their advantages and depending on the application, the Eulerian framework might be better suited.

[KK07] state that for the Lagrangian framework the inertia of objects is exploited. That is because the vertex positions between frames are usually linearly interpolated in order to find their position for arbitrary samples in the shutter interval. Consequently, one assumes constant velocity for the objects and the error of this estimation is proportional to the acceleration. This often works quite well in practice.

Assuming that the a simulation in the Eulerian framework resulted in a grid of data values for each frame, they show that linear interpolation of these values (analogous to the Lagrangian framework) produces incorrect results. The error introduced by interpolating the density (in the case of smoke) or level-set (in the case of water) values even is not related to the grid resolution. Instead they propose to estimate these values for arbitrary time samples by exploiting the inertial movement of the fluid by advecting the values along the velocity field.

Thus for each grid cell that is traversed by the ray they determine its value as follows. They use the velocity field at the current cell to compute a back-tracked position (by particle tracing methods) similar to [Sta99]. They then use the value of the grid cell at this position to get an estimation of the advected value. Unfortunately the velocity field is needed to compute advection which leads to a larger memory footprint during rendering compared to the simple interpolation of grid values.

3.2 Analytic Computation

As stated above, Monte Carlo methods might need a lot of samples for a good estimation of the pixel integral if the variance of L is large. Fast movements of objects are such a source of high variance.

In a production environment, the shading function of objects is quite arbitrary. However the visibility function is well defined by the geometry and the corresponding motion paths. Instead of solving for both the visibility and the shading function stochastically over time

(as done when using the Monte Carlo integration approach above), we now try to solve the visibility function analytically while keeping the stochastic integration for the shading.

Thus, analytic visibility computations, such as the method described in [KB83], reduce variance by effectively removing the time dimension of Equation 3.1. This is achieved by computing the visibility interval T_k for each object which is visible for the current pixel during the shutter interval. Consequently, the cost for taking samples in increased – yet if the variance reduction is large enough, the efficiency of the Monte Carlo estimator increases.

Let $L_k(\mathbf{x}, t)$ denote the radiance that is emitted from the k-th primitive in the scene into the direction of $\mathbf{x}$ on the image plane. By introducing the visibility function $V_k(\mathbf{x}, t)$ that equals one whenever the k-th object is visible and is zero otherwise, we can rewrite Equation 3.1 as follows:

$$I = \int_A \int_T \sum_k V_k(\mathbf{x}, t) L_k(\mathbf{x}, t) dt d\mathbf{x} \tag{3.3}$$

$$= \int_A \sum_k \int_{T_k} L_k(\mathbf{x}, t) dt d\mathbf{x}. \tag{3.4}$$

By setting

$$L_k(\mathbf{x}, T_k) := \int_{T_k} L_k(\mathbf{x}, t) dt \tag{3.5}$$

$$\approx \frac{|T_k|}{M_k} \sum_{i=0}^{M_k - 1} L_k(\mathbf{x}, t_i) =: \hat{L}_k(T_k) \tag{3.6}$$

for appropriate integers M_k (e.g. dependent on $|T_k|$) and samples $t_i \in_R T_k$ we get

$$I = \int_A \sum_k L_k(\mathbf{x}, T_k) d\mathbf{x} \tag{3.7}$$

$$\approx \int_A \sum_k \hat{L}_k(\mathbf{x}, T_k) d\mathbf{x} \tag{3.8}$$

$$\approx \frac{|A|}{N} \sum_{i=0}^{N-1} \sum_k \hat{L}_k(\mathbf{x}_i, T_k). \tag{3.9}$$

Usually the number of samples M_k does not need to be large since shading changes during the shutter interval often have low variance. Note that no primary rays need to be shot for estimating $L_k(\mathbf{x}, T_k)$ since the visibility is already known. In an implementation, the sum over primitive indices would obviously only run over those k with $T_k \neq \emptyset$.

The difficulty here is determining the non-empty visibility intervals T_k. Usually we have to restrict ourselves to certain kind of primitives and simple motion to efficiently compute these intervals. For triangles and piecewise linear motion segments such an approach is described below.

There have been attempts at solving for both the temporal visibility and pixel coverage analytically at the same time [Gra85]. However, as mentioned by [SPW02] this approach suffers from precision problems since the 4D-representation of 3D-polyhedra in linear motion used by Grant generally is only an approximation.

We would like to note that this algorithm can also be used to generate interval photons, as described in Subsection 3.1.5. When this approach is used for rays originating at the camera as well as for photons, the second pass of the photon map rendering algorithm needs to locate those photons whose time intervals overlap with the visibility intervals. The intersection of these two intervals then describes the valid interval for such a photon.

3.2.1 Triangles and Linear Motion Segments

Besides restricting the movement of objects to linear segments we additionally require the geometry to stay "topologically equivalent" during motion [AGP+95, p. 90] for this algorithm. Together with some heuristics for shading a similar analytic algorithm has been used in the Maya renderer [SPW02, PS01].

For each ray we need to find all candidate triangles that might intersect during their movement. In a pre-processing step Korein, Pearce et al. therefore project the convex hull of the moving triangle onto the image plane. After rasterizing the convex hulls they know for each pixel which triangles might intersect a given primary ray. However this only works for primary rays, not for secondary effects. For a global illumination renderer it is impractical to rasterize all convex hulls for each ray origin.

That is why we propose to build an acceleration structure over the convex hulls (or their axis-aligned bounding boxes) in a pre-processing step. By traversing this structure we can get the candidate triangles for arbitrary rays, including all secondary effects or photon mapping algorithms [Jen01].

A certain disadvantage of this approach is the increased complexity of simulating motion blur caused by camera movement. Instead of moving the camera stochastically we now need to apply the inverse motion to all objects. This might cause acceleration structures to degrade since the bounding boxes of the objects have to be enlarged. Tilting the camera can have immense effects on axis-aligned bounding boxes (similar to effects known for the rotated "kitchen" scene of the BART benchmark [LAM01]).

After the candidate triangles have been found we need to determine the corresponding visibility interval for each such triangle. We consider each triangle edge separately. Since we constrained the motion of each triangle vertex to a linear segment (of course complex motions can be split up into a number of linear segments) each moving triangle edge forms a bilinear patch (see Figure 3.1).

To determine the full interval, we begin by intersecting the ray with the triangle for the start of the shutter interval. Then we intersect the three bilinear patches related to the triangle edges. Each time an edge is intersected, the corresponding visibility interval either starts or ends, based on the state before the intersection. When intersecting the bilinear patches, care has to be taken when solving the quadratic equation in a numerically stable way [KK89, RPH04].

The algorithm can be outlined in two phases:

- Pre-processing:
    1. Approximate motion by linear segments.

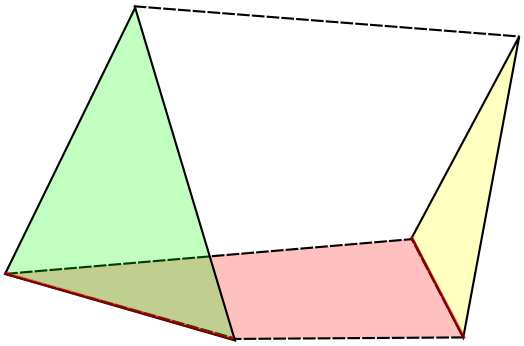


Figure 3.1: During the shutter interval, each of the vertices of the left (green) triangle is transformed using a linear segment. At the end of the shutter interval, we reach the right (yellow) triangle. The surface described by each edge of the triangle (e.g. the red one) forms a bilinear patch. Note that this patch usually is not planar.

2. Build an acceleration structure over the bounding boxes of the moving triangles (containing the triangle during the entire shutter interval).

- Rendering:

1. For each eye ray, use the acceleration structure to intersect the ray with all moving triangles along the ray. For each triangle, three bilinear patches have to be intersected with the ray.
2. Clip the resulting time-segments against each other and compute mutual occlusion.
3. Use the length of the resulting non-overlapping time-segments to choose and weight shading samples.

After the moving triangles along the ray have been found, the visibility intervals need to be clipped against each other to account for mutual occlusion. This works analogously to the Watkins visibility algorithm [Wat70] for a scan line. The x-coordinates along the scan line now correspond to t-coordinates along the shutter interval (see Figure 3.2). Given m intervals as input, the worst-case complexity of this algorithm is $\mathcal{O}\left(m^2\right)$ since there are at most $\mathcal{O}\left(m^2\right)$ scan line events where the visible segment changes (either endpoints of segments or intersection points of segment pairs).

With the final set of non-occluding segments the only variance that is left now is due to shading changes during the visibility intervals. Usually this variance is rather small (that is why RenderMan only shades at endpoints of linear motion segments, not in between). It therefore seems sensible to select the number of samples to take according to the length of the visibility intervals. The resulting estimations for each object are weighted according to their contribution relative to the complete shutter interval and summed up.

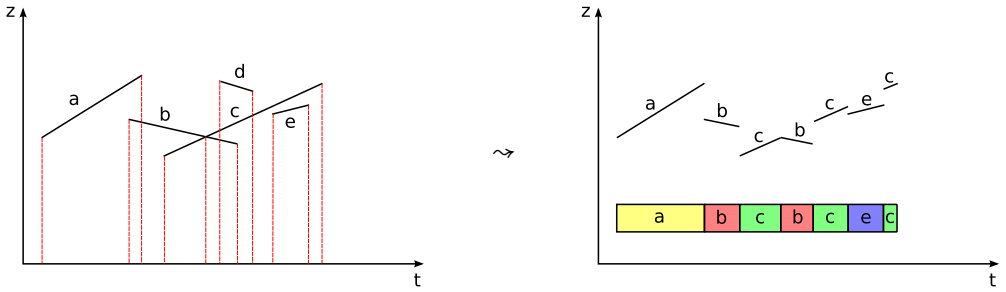


Figure 3.2: The Watkins algorithm [Wat70] determines the visible segments according to mutual occlusion. Possible events along the scan line where the visible segment may change are denoted with red, dashed lines. The x-coordinates along the scan line now corresponds to points of time t. The resulting decomposition of the shutter interval is shown on the right.

Early Out Culling

The classical traversal of acceleration hierarchies that do not account for moving objects is as follows: A ray $r(t) = \mathbf{o} + t \cdot \mathbf{d}$ with origin $\mathbf{o}$ and direction $\mathbf{d}$ is given an initial valid parametric distance range $t \in [t_{\text{min}}, t_{\text{max}}]$. During traversal, the t_{max} parameter is decreased to the value t_{hit} whenever an intersection is found for the ray parameter $t_{\text{hit}} \in [t_{\text{min}}, t_{\text{max}}]$. This improves performance since parts of the acceleration structure lying "behind" the hit point do not need to be traversed (also known as "early out" culling). This yields correct results since we are interested in the *first* hit point along the ray.

In the case of intersecting linearly moving triangles, the singular parametric t_{hit} values above become intervals. We propose to combine the traversal and clipping steps in the algorithm outline: We clip each segment against all previously found segments directly after it has been found. Each of these clipped segments has a corresponding parametric interval $[t_{\text{hit-min}}, t_{\text{hit-max}}]$ for the ray intersection. It is safe to set the ray interval end t_{max} to the *maximum* of these parametric segment intervals for early out culling (with the restriction that t_{max} must not be updated when there exist parts of the shutter interval with no corresponding segments). The usage of early out culling can reduce traversal complexity significantly for scenes with large depth-complexity.

Memory Requirements

The bilinear patches do not have to be stored explicitly. Instead, we can interpret the triangle edges as bilinear patches on-the-fly. However, we cannot reuse acceleration hierarchies for instances anymore, since the bounding boxes of the triangles might differ for each instance due to different movements. This might cause increased memory consumption when using instanced geometry.

Conclusion

We believe that for most scenes this algorithm is worth the increased complexity for taking samples since one dimension of the integration problem is removed [SPW02]. Usually this temporal dimension is responsible for large variance and results in noisy images. When using the analytic solution we only need additional samples for spatial aliasing and to account for shading changes. Visibility intervals are one of the very few things that can be computed analytically with manageable effort in realistic image synthesis. It therefore is rather surprising that none of the production renderers available at the moment of writing employ this approach.

It is furthermore possible to clamp the shading frequency as successfully employed in Pixar's PhotoRealistic RenderMan. One could evaluate the object's shading for a small number of fixed points in time of the visibility interval (e.g. at the beginning and the end). These results are then interpolated to get the shading for arbitrary points in time.

This can be combined with a caching strategy: These fixed shading points are only evaluated whenever they are needed for the first time. To reduce memory footprint, results that have not been used for a long time can be discarded. This also solves the problem of earlier RenderMan implementations where many shading results were not used since the shading took place before visibility computation [Cat84].

Of course it is desirable to directly solve for temporal coverage regarding the pixel area (similar to [Gra85], but without approximations). However we expect this to be computationally much more expensive and the problem of integrating the shading function over time remains. Thus we believe that both spatial anti-aliasing and shading integration is handled best by Monte Carlo approaches.

While analytic visibility computation is certainly very useful, at some point analytical solutions are not available anymore, for reasons including

- arbitrary complex shading functions, making their analytical integration impossible,
- high-dimensional integrals in order to incorporate anti-aliasing, deph of field, participating media, global illumination, etc.,
- complex filter functions beyond simple box-filtering, making analytical solutions either impossible or extremely expensive.

Therefore at least some parts of the rendering pipeline need to use sampling methods. Low-discrepancy points that are well suited for such purposes are described in Chapter 5.

Motion Blurred Lines on the GPU

The problem of drawing linearly moving two-dimensional lines with a certain thickness can be considered a special case of the algorithm above. One could send a quadrilateral the size of the convex hull of the moving line on the GPU. For each fragment, the coverage then can be computed analytically in a shader by intersecting the fragment position with the four sides of the moving thick line.

If the line has a solid color that changes linearly it is possible to integrate the shading analytically, too. Instead of using expensive multi-sampling to avoid jaggies at the start- and end-positions of the line, an approach similar to [CD05] could be used.

3.3 Approximate Methods

Both the Monte Carlo approach and the analytic visibility computation described above produce correct results for valid input data. In contrast to this a number of approximate methods have been developed. These algorithms often work extremely fast, yet they produce convincing results only for some settings.

3.3.1 Adapting Geometry

The idea here is to transform the moving geometry of the scene to static geometry that captures the effect of movement, actually pre-filtering in object-space prior to rendering [IK00]. This makes building acceleration hierarchies for ray tracing much easier and reduces variance extremely due to the removal of the temporal dimension in the pixel integral.

[WZ96] presents a real-time motion blur approximation that sweeps surfaces along their motion path and renders them semi-transparently. Each object is split up into a front and back part. Matching joining parts corresponding to the motion vector of the object are constructed afterwards. The algorithm is not able to handle polyhedra of non-zero genus (e.g. a torus) realistically and cannot address inter-object relations correctly. However it is very fast and utilizes the acceleration abilities of the graphics card.

Very similar in nature is the approach by [JK05]. They try to find the silhouettes of motion of meshes and decrease the opacity along the extruded surfaces (called the blur shell). Their algorithm is not meant as a physically correct simulation, but instead as a real-time artistic method for motion-blur.

It is not clear how to generate sweep-surfaces for arbitrary primitives and movement efficiently. Even more difficult is finding appropriate modified material properties for the primitives to capture the effect of motion blur correctly. Usually the objects have to appear transparent depending on their movement speed relative to the image plane. Yet the temporal change in illumination around the moving object is very difficult to capture using modified shading functions. For special cases such as moving spheres in [TBI03] or interactive applications where crude approximations are sufficient this may be a valuable approach nevertheless.

[Cat84] transforms temporal contribution to spatial contribution as well. However there are problems when blurred polygons intersect non-moving polygons. Furthermore it is not clear how to handle transparency and secondary effects, including shadows.

[GM04] presents a method suitable for point-based surface rendering. They extrude the circular points used to define the surface into ellipsoids and render them using hardware acceleration. However the method is meant only as rendering a motion hint, since they do not blur the moving object. Instead they use textures to generate artistic (and possibly non-photorealistic) effects.

3.3.2 Clamping the Shading Frequency

Pixar's PhotoRealistic RenderMan [CCC87] clamps the frequency of the shading function by precomputing shaded results at a fixed number of points in time. Usually there is only one shading result at the beginning and the end of the shutter interval. By the use of multiple linear motion segments, the number of shading results can be increased in newer versions of PhotoRealistic RenderMan, though.

The reasoning for doing this is the complexity of current shading functions. Compared to shooting rays it usually is much more expensive to evaluate a shader function for typical movie production renderings. By decreasing the number of shader evaluations, the total rendering time is decreased enormously, outweighting the obvious artifacts (e.g. highlights on moving objects can often be captured only poorly) that are inherent of this method. This approach can be combined with Analytic Computation.

Earlier versions of PhotoRealistic RenderMan could only render motion blur of the moving objects themselves, not of shadows or reflections. The addition of pre-filtered deep shadow maps [LV00] later allowed for this. The ray tracing based RenderMan implementation Blue Moon Rendering Tools (BMRT) [GH96] and its successor ExLuna both were able to render motion blurred shadows and reflections [AGP+95].

3.3.3 Post-Processing

There are many approaches that try to simulate motion blur after the rendering has taken place, i.e. using only two-dimensional images. In general there are many scenes where all such post-processing methods are doomed to fail [CPC84, AGP+95]. However for fast approximations some of these algorithms might be suited very well.

One of the earliest attempts is by [PC83] who produced motion blur by convolving images of objects with the motion function using fast Fourier transformations. To handle multiple objects with different motion paths (or still backgrounds), multiple images are processed and blended together. However this does not allow for visibility changes within a single image and gets quite expensive with large numbers of objects.

Alternatively, one can compute the optical flow field amongst the images and blur appropriate areas. This is especially useful whenever the motion function is not known. For stop-motion animation this approach has been used successfully [BE01]. If the trajectories of the objects are known (e.g. when the images are generated by a ray tracer), the optical flow field can be used to capture indirect effects such as moving shadows and reflections. [ZKTR06] therefore combines both the available motion information and the optical flow to yield better approximations. However optical flow algorithms have problems when there are significant intensity changes between frames. Moreover object boundaries might be smeared out due to the smooth optical flow fields.

The pixel-tracing filter [Shi93] is a post-processing implementation that tries to trace pixels corresponding to the same object. It computes a velocity field built be extracting sub-pixel information from image sequences. Its appeal lies in the independence of scene complexity and moderate computational cost. The algorithm has been extended in [Shi95] to handle motion blurred reflection, refraction and shadows as well. However the memory requirements

are quite high, since the complete ray tree for each pixel for many subsequent frames needs to be stored.

[ML85] present a $2\frac{1}{2}$D algorithm that takes input images for each object, applies a motion blur approximation, sorts the objects in depth and finally composits these images into a single picture according to depth order. This idea has been extended in [Max90] to handle polygonal objects where each vertex may move independently, allowing for deforming objects instead of forcing the same motion vectors to be valid for all vertices.

The Pixmotor algorithm [Neu07] is an image-based approach as well. For each pixel it takes a motion vector and a depth channel to generate a motion blur approximation. Each pixel is moved along its linear trajectory and the result is accumulated while a z-buffer is used to handle occlusion. Since only a single motion vector and color value is stored per pixel, the algorithm cannot handle transparent objects correctly. Neulander describes several heuristics that are used to handle "holes" in the accumulated image that are results of the absence of relevant hidden surface information in the input image.

A post-processing technique that is very well suited for GPUs is presented in [Ros07]. To achieve high frame rates it may be too expensive to send the scene data more than once to the graphics card to perform multiple passes in order to approximate motion blur. Therefore Rosado's GPU shader uses the depth buffer and the inverse view-projection matrix to recover the world-space position of each pixel. This world-space position is then used together with the view-projection matrix of the previous frame to compute a velocity vector. Finally the color of the current pixel is determined by averaging multiple samples along the direction of the velocity vector. While this works rather well for camera movements, velocity fields for dynamic objects must be computed separately.

Animations are often rendered frame by frame, however subsequent images of such a sequence are likely to be very similar. [HDMS03] builds upon this observation using a multi-frame ray tracer that exploits temporal coherence by recycling samples. They implement motion blur by tracking intersection points across frames through reprojection and distribute incoming radiance amongst pixels using a DDA line algorithm [SAG+05].

3.3.4 Predictor-Corrector Methods

For many scenes two-dimensional post-processing concepts achieve very convincing results. However in some cases they generate incorrect images. The idea behind the following algorithm is to combine the best of two worlds: The speed of image-based algorithms, and the correctness and generality of stochastic algorithms that operate in 3-dimensional space.

We first partition the shutter interval $[t_0, t_1)$ into N smaller intervals of length $\Delta t := \frac{t_1 - t_0}{N}$, denoted $i_k := [t_0 + k\,\Delta t, t_0 + (k+1)\Delta t)$, where $k = 0, \ldots, N-1$. It is clear that when the N images for all intervals i_k are computed and averaged, the result is equal to using the full shutter interval. The algorithm starts with computing images for the first and last interval i_0 and the last interval i_{N-1}. Afterwards, the algorithm continues recursively to generate the remaining images for the intervals $i_1, \ldots, i_{N-2}$ (see Figure 3.3). This recursion scheme works without rounding problems for $N = 2^m + 1$, where m is a positive integer. This can be proven easily by induction.

1. Given images for intervals i_j and i_k, with $j < \frac{j+k}{2} < k$, *predict* the image for the interval $i_{\frac{j+k}{2}}$ using the given images. This prediction is computed in image-space.

2. Determine where the image-based prediction failed and *correct* these regions using the stochastic approach as a fallback method.

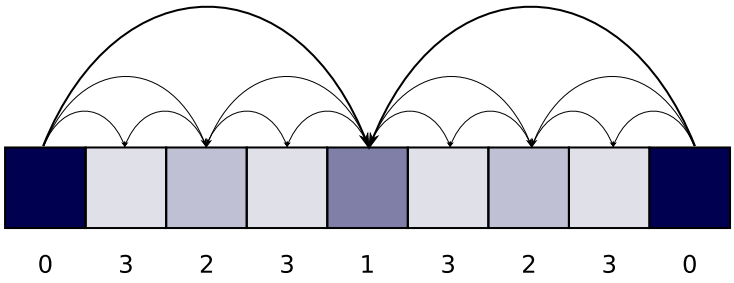


Figure 3.3: Diagram of the hierarchical structure. The corresponding images for the leftmost and rightmost intervals are used to predict the image for the interval at the middle. This image is used afterwards to predict two other images, and so on. The numbers at the bottom show the recursion depth.

This approach is inspired by Keller's hierarchical Monte Carlo method to compute a scan line of an image [Kel01]. In that context, a single pixel of the scan line corresponds to one of the images for an interval i_k above. Keller achieves an unbiased result by using the same set of samples for each pixel and using a predictor that interpolates linearly. This way, a separation of the main part (also known as a control variate) can be applied, resulting in much reduced variance.

While this works well for scan lines, the approach cannot be directly transferred to the context of motion-blur, as shown in Figure 3.4 and Figure 3.5. We therefore apply a nonlinear prediction and use different samples for each pixel and interval. Please note that the resulting image therefore is no longer unbiased.

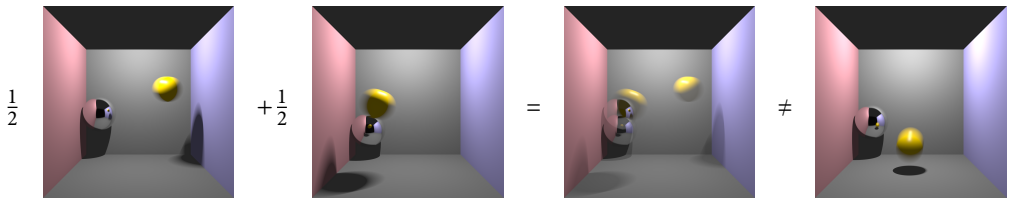


Figure 3.4: Linearly blending over images at the start and end of the shutter interval yields a poor approximation for the middle of the shutter interval, depicted on the right.

The reasoning behind the hierarchical computation is still valid, though. For most scenes, the lower the level of the recursion in the hierarchical computation, the better the prediction on this level will be. That is because the distance of intervals i_k and i_j used for prediction

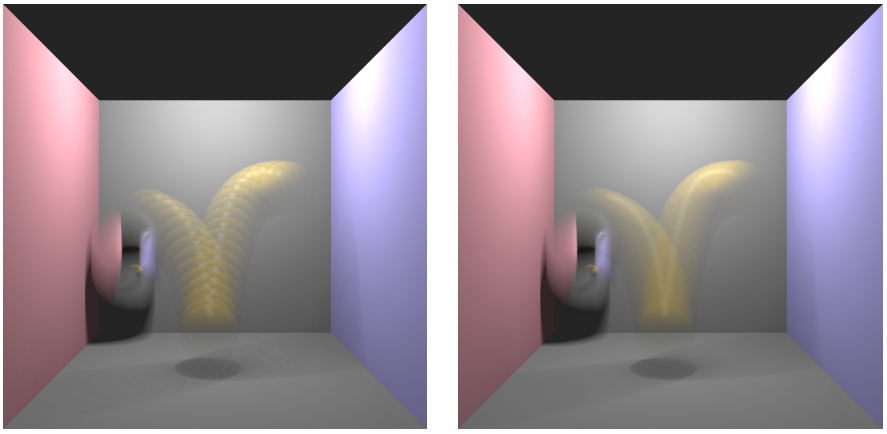


Figure 3.5: On the left, the same 32 samples are used per pixel. This yields the same result as blending over 32 images rendered at distinct points in time. On the right, the same sample set was used per pixel, but each with a random Cranley-Patterson rotation. The resulting image is noisy but visually much more appealing.

decreases on lower levels (see Figure 3.3). In other words, for most scenes the images corresponding to i_j and i_k get more similar the smaller $|j - k|$ is. Since the prediction gets better as well, the effort for correcting the image can be decreased.

Nonlinear Prediction

When rendering the images used for the prediction, we store the following information for each pixel in addition to color (using a "deep" framebuffer):

- t-parameter along the ray (corresponds to a depth buffer),
- 3-dimensional hitpoint with inverse object transformation applied,
- time coordinate of the sample,
- primitive identifier.

Since we take numerous samples per pixel, we have to decide for which sample we store this information. The choice is rather arbitrary, but we decided to store this information whenever an object gets hit that lies nearer to the camera eyepoint than the previous one. We therefore initialize the framebuffer depth-values with ∞. This approach favors objects in the foreground over background objects, the latter often being static.

Using this information, the prediction step is rather straightforward. We interpret four adjacent pixels as two triangles (see Figure 3.6) and transform the vertices to the new time interval by applying the corresponding object transformation to the vertices. For that purpose we simply add the constant offset $\frac{k-j}{2}\Delta t$ to the time coordinate of the sample. Since we applied the inverse transformation to to the stored hitpoint, this transforms the sample in a correct way. Now the 3-dimensional coordinates get reprojected onto the image plane

and the resulting triangles are rasterized. The rasterizer interpolates color linearly and only overwrites pixels when their depth values are larger (similar to the classic z-buffer).

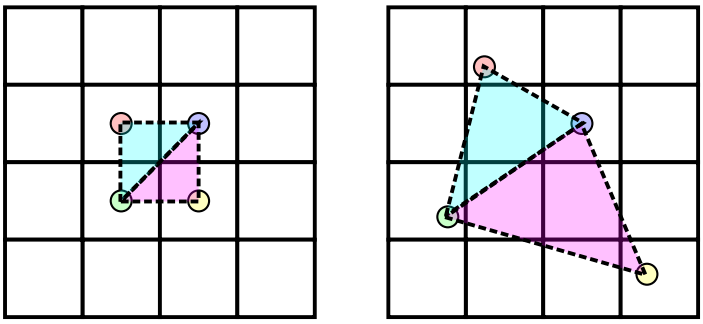


Figure 3.6: The predictor takes each four adjacent pixels, transforms the stored sample values to the new time interval, and finally rasterizes the resulting triangles. It interpolates the previous vertex colors when doing so.

We create two such predictions, one for the interval i_j and one for the interval i_k. We now need to mix these predictions to create our final prediction for $i_{\frac{j+k}{2}}$. We average the pixel colors and could randomly decide which of the remaining information we should take. Instead we have implemented a deterministic approach depicted in Figure 3.7.

0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0

Figure 3.7: We create two predictions for the image corresponding to the interval $i_{\frac{j+k}{2}}$: one for i_j and one for i_k. The resulting colors are averaged, but the remaining information, such as the instance that was hit, cannot be interpolated. We therefore alternate in our choice using the parity of a binary exclusive-or operation of the pixel coordinates.

We would like to stress that the complexity of the prediction step is completely independent of the scene rendered and the numbers of samples taken. Instead, its complexity depends linearly on image resolution if we assume that the time spent for rasterizing a triangle is bounded by a constant.

While we have implemented the rasterization in software, this step could take advantage of the extreme parallelism of current graphics cards. Every four adjacent pixels could be sent as a triangle strip to the graphics pipeline. We only need a linear interpolation of the vertex colors (equal to the original pixel colors) and no lighting or fragment shader programs. This promises extreme performance. Furthermore, OpenGL extensions to render directly to

a texture could reduce memory bandwith. Care must be taken to retain precision of the intermediate results. Most modern graphics card offer 32-bit floating point support, which should be sufficient.

Prediction by Optical Flow

The nonlinear prediction described above handles moving objects rather well, but does not predict movement of secondary effects like moving shadows and reflections. The shading changes of a surface that mirrors moving object cannot be handled correctly since the surface itself does not move. We believe this could be improved by using optical flow algorithms to track the motion paths of these effects between frames as used in [BE01]. Using both the available motion information as well as optical flow seems promising as well [ZKTR06].

Implementations of the Kanade-Lucas-Tomasi (KLT) Feature Tracker [LK81, TK91, ST93] to generate optical flow fields are available as open source as well as in Intel's OpenCV library. The "GPU_KLT" implementation furthermore utilizes the graphics card to accelerate the tracking. Correct tracking of soft shadows and blurry reflections might remain difficult due to the absence of large contrasts. Optical flow algorithms that rely on edge detection or segmenting are likely to yield unsatisfactory results then.

Stochastic Correction

We need to detect those cases where the predicted value differs by a large amount to the true value. To estimate the correct value, we can use N_l random samples that should get smaller as the level l gets lower (since we assume that the prediction gets better as well). As mentioned in [Kel01, Sec. 3.3], this direct estimation should not be compared directly to the prediction, since the number N_l usually is too small to fulfill the assumptions of the law of large numbers. Instead a contrast criterion similar to the one used by Keller is more appropriate. Whenever such a contrast measure exceeds a certain threshold we discard the prediction and use additional $N_{\text{fail}} - N_l$ samples to get a better estimate of the true value. We also use N_{fail} samples directly for those border regions of the image where no triangle of the prediction step (see Figure 3.6) got rendered.

The stochastic part of the algorithm needs to shoot rays through the scene, employing an acceleration structure. This structure is likely to be much more efficient if rebuilt for each of the N images (also see Chapter 4). Bounding boxes for moving objects that include the movement over the time interval are very likely to be smaller, since we are not considering the whole shutter interval $[t_0, t_1)$, but only an interval of length Δt.

Since we need to store N potentially very large images, the algorithm can also operate on image tiles to reduce the memory usage. After the result of a tile is computed, the temporary results can be used to compute the next tile.

3.3.5 Depth of Field by Motion Blur

Whenever non-pinhole cameras with non-zero lens apertures are simulated, some parts of the scene are out of focus, depending on their distance to the lens. Rendering models for

simulating realistic lens models can be found in [KMH95, HSS97]. A survey on methods to simulate depth of field using a graphics card can be found in [Dem04].

It is possible to simulate depth of field effects of a thin lens approximation using the analytic motion blur approach presented above. The basic idea is very similar to the one presented in [AMMH07]. Instead of using discrete sample points on the lens, and averaging images rendered using such camera shifts in an accumulation buffer [HA90] as shown in Figure 3.8, we can average images rendered corresponding to sampling *lines* on the lens (see Figure 3.9). As noted by [KMH95] and [HSS97, Sec. 3.5], such a simple averaging scheme is only an approximation since the correct exposure of the film is not accounted for. Most camera models used in computer graphics suffer from this, though.

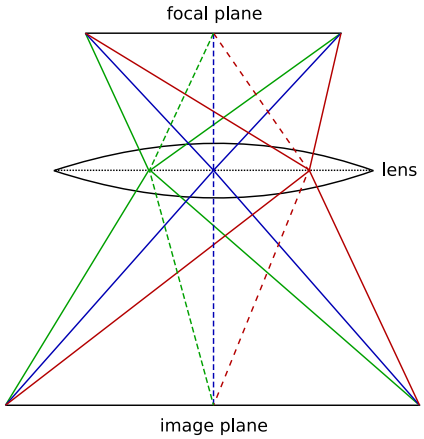


Figure 3.8: Each two-dimensional slice of the four-dimensional light field can be computed using a standard perspective projection [HSS97].

Instead of adjusting the camera we apply the inverse transform to the geometry and render one motion blurred image for each sampling line. Unfortunately we cannot combine motion blur and depth of field, since the movements due to the sampling on the lens and the objects are then correlated in time.

Compared to uniform sampling using discrete points on the disk, it is obvious that the sample density is much higher at the center of the disk when using the lines shown in Figure 3.9. It is therefore important to redistribute the samples as depicted in Figure 3.10.

To achieve the effect of moving with increasing speed along the line, the nearer the current point is at the center of the lens, we need to modify the samples taken after the mutual occlusion has been computed. Instead of using uniformly distributed samples, we need to remap them using a $\sqrt{t}$-like function. The resulting images are shown in Figure 3.11.

We cannot use the same acceleration structure for secondary effects since shadow and reflection rays should stay unaffected by depth of field. Therefore we need to build a second acceleration hierarchy without the movement originating from the depth of field simulation. Alternatively, we could use deferred shading [ST90] and build this second hierarchy after all primary rays have been shot.

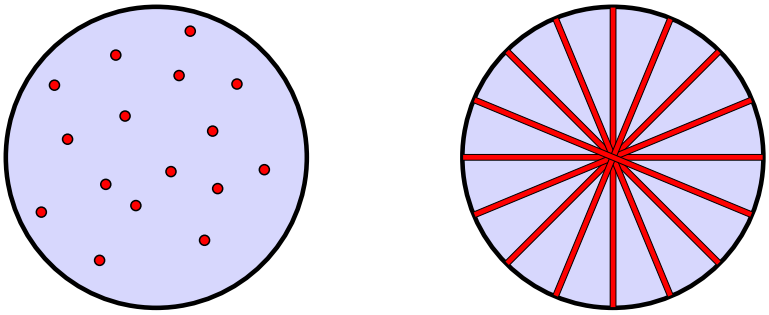


Figure 3.9: Instead of averaging images rendered from discrete points on the lens aperture, one can average images corresponding to sampling lines on the lens. The line pattern on the right features long lines and a large number of different angles to avoid banding artifacts.

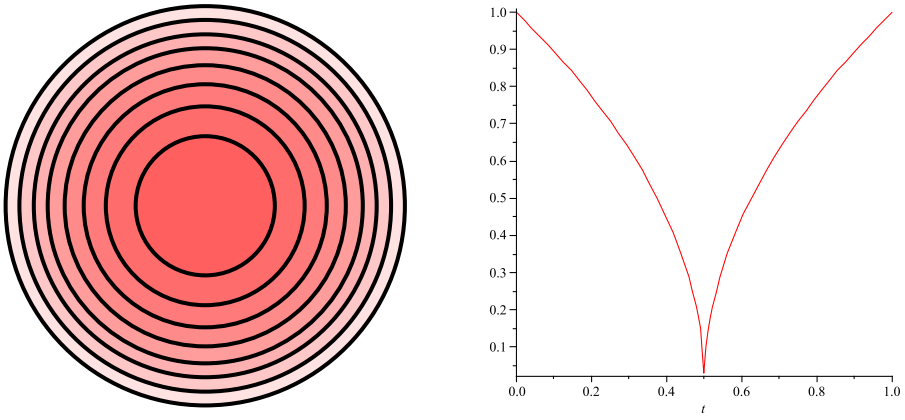


Figure 3.10: Care must be taken to approximate the uniform density of the sample points on the lens aperture correctly. A $\sqrt{t}$-mapping (shown on the right) must be applied to redistribute the samples.

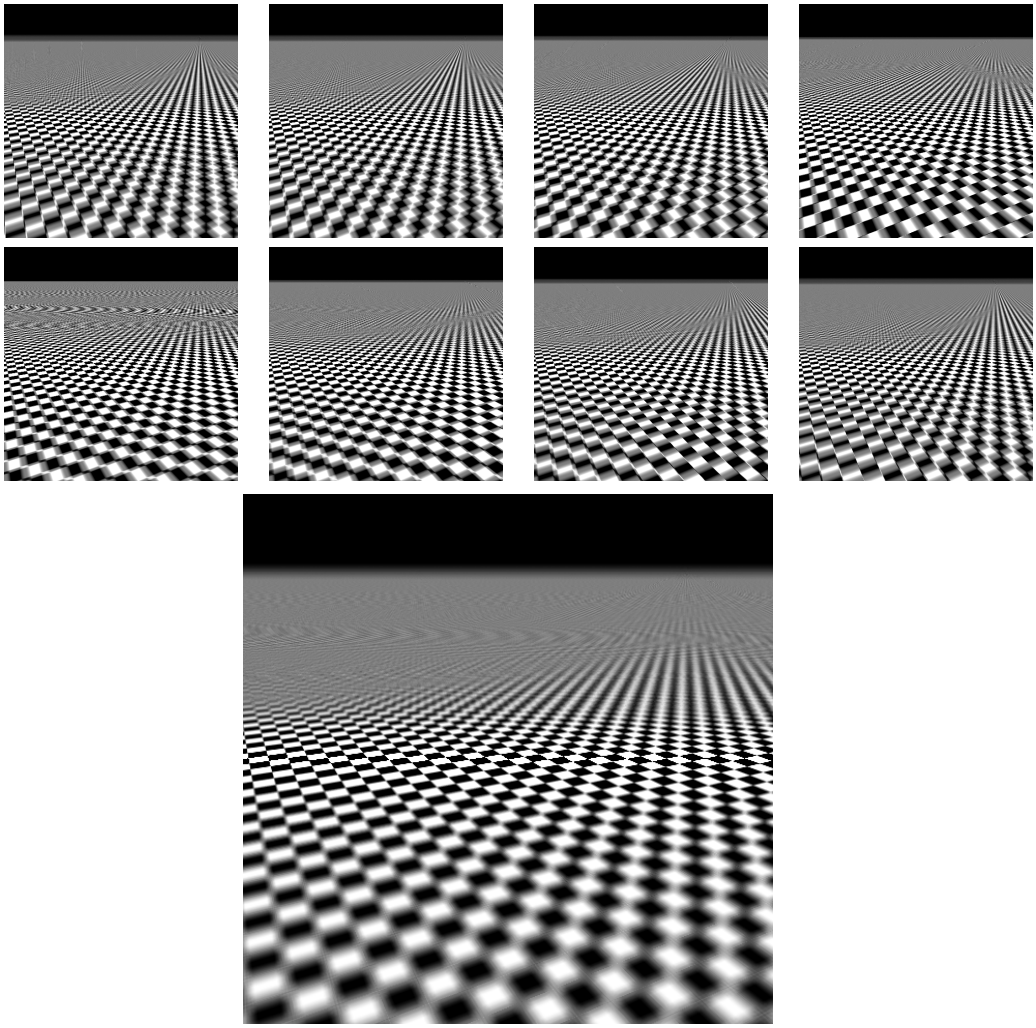


Figure 3.11: Averaging the eight motion blurred images at the top yields the depth of field approximation at the bottom. When 16 images are averaged, no banding artifacts are noticeable anymore for this scene.

4 Ray Tracing in the Presence of Motion

At the core of every ray tracing architecture lies an acceleration structure that allows for the fast processing of ray queries. This chapter describes methods to improve such structures, paying special attention to the problem of extending those structures to the time dimension.

4.1 The Surface Area Heuristic (SAH) for Partitioning

The surface area heuristic [GS87] is the currently best known method to guide the partitioning step during the build of acceleration structures. It can be used both for partitioning object lists (to build BVHs) and to partition space (to build kd-trees). In spite of its widespread use, it is still not completely understood why the SAH works so well.

The local heuristic is usually derived using conditional probabilities [GS87, Hav00]. However, trees built using the SAH perform exceptionally well even if the assumption of uniform ray density is not justified at all. For example this is the case for primary rays with a common origin. [BH07] estimated the probability that a ray traverses a bounding box of a node and used this information to build another tree. If the surface area heuristic derivation was justified, one would expect this to pay off. Instead, Bittner and Havran reported only minimal improvements or even degrading performance. An additional oddity is the linear weighting of the surface areas using the number of primitives, when in fact we expect the tree to have logarithmic height [WH06].

Despite all this, even slight modifications of the heuristic usually result in deteriorated performance for most scenes. That is why we conclude that the SAH actually performs well because of reasons quite different from the original derivation.

4.1.1 Interpretation of the SAH

When all common factors and constants are removed from the SAH cost term [Wal07], the following expression is evaluated for each axis and possible j-split position:

$$c(j) = N_{L,j}A_{L,j} + N_{R,j}A_{R,j}, \tag{4.1}$$

where $N_{L,j}$ and $N_{R,j}$ denote the number of primitives for the left and right child for the j-th split position. $A_{L,j}$ and $A_{R,j}$ denote the surface area of the bounding boxes that encloses the primitives of the left and right child. The split position is determined by minimizing $c(j)$.

It is generally accepted that for good bounding volume hierarchies (BVH) the summed volume of all nodes is minimal [KK86]. This is similar to maximizing the "free" space between nodes and minimizing overlap of nodes. These criterions have all been successfully applied for R*-trees [BKSS90]. R-trees (respectively Boxtrees with very similar structure) have been mentioned previously in the context of ray tracing, e.g. in [WBS07, Bez01, GWH01, YCM07].

In the context of minimizing summed volume, the cost function $c(j)$ can be interpreted in a different way. For a fixed split position j, the summed area of the bounding boxes of all nodes in the current subtree can be bounded. In the worst-case we only split one triangle for each level of the remaining subtree and each triangle has the same bounding box as the bounding box of its parent node. Then the height of the tree is limited by the number of primitives in the node that gets split in such a worst-case fashion. Thus, an upper bound of the summed area of all nodes of the current subtree is given by

$$s(j) = A + (2N_{L,j} - 1)A_{L,j} + (2N_{R,j} - 1)A_{R,j}. \tag{4.2}$$

When we neglect constants and common factors it is clear that $c(j)$ and $s(j)$ get minimized for exactly the same j. Hence the SAH can be interpreted as minimizing the summed area of all nodes of the tree.

Intuitively, one might want to minimize summed volume instead of area. However some objects might have bounding boxes with zero volume (most notably triangles that lie in a plane perpendicular to one of the cardinal axes). Using the volume criterion, triangles whose common bounding box has zero volume cannot be splitted effectively. This might be the reason why the area measure works better for most scenes. We would like to note that the sum of volume and area yielded superior results for some scenes.

This does not explain the termination criterion of the SAH, though. However for some scenes performance is actually increased by not using this criterion. Instead a leaf is created whenever the number of primitives falls below an arbitrary constant or the tree gets too deep.

If one tries to minimize the summed area of the two child nodes locally without using the factors $N_{L,j}$ and $N_{R,j}$, the tree gets very deep and inefficient. That is because one does not account for the lower levels of the tree. The weighting causes the SAH cost function to be shaped similar to a parabola, since both the area and the number of primitives are approximately linear along the splitting axis. This means the SAH favors splits that evenly divide the number of primitives to both children, except when there are jumps in the surface areas of the bounding boxes. Together this helps to cut away free space while keeping the tree depth low.

4.1.2 Optimal Local Subtrees

We fix the number m of primitives for which we can generate an optimal tree in the sense of minimal summed area. For every largest subtree of the SAH guide tree not containing more than m primitives, we generate an optimal tree and treat the bounding box of this whole tree as a new primitive. We then run the SAH algorithm again, however with these new primitives (essentially building a tree over trees). This scheme is iterated until we have rebuilt the complete tree. The resulting tree is guaranteed to have no larger summed area than the original SAH tree. Usually the summed area should decrease, resulting in increased performance.

The optimal tree for up to m primitives can be generated using a backtracking scheme. We terminate the recursion whenever the current summed area exceeds the best "known" value. This value is initialized with the result of the previous SAH tree.

SAH binning approaches could be used to speed up the repeated construction of the SAH tree [Wal07].

Alternatively, after having built the tree once using the SAH, we can visit each level of the tree, starting at the bottom. We then pick all nodes exactly k levels deeper (i.e. up to 2^k nodes), if exisiting. These primitives are represented by bounding boxes. We then search for the tree that has minimal summed area for these primitives. After having found this tree, we exchange the current subtree with the optimal one, reestablishing all links to deeper levels that were present before. After having done this for all subtrees of this level (which can be parallized easily, since the subtrees are non-overlapping), we proceed to the next level above, until we reach the root.

To find the minimal summed area table for m primitives as leaves, we can consider all *extended* binary trees [Knu97, Sec. 2.3.4.5], as explained in [Pro80]. Thus we need to generate all possible binary trees with $n = m - 1$ nodes. The number of different binary trees with n nodes is given by the Catalan number

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}. \tag{4.3}$$

Since for all these trees different permutations of the $m = n + 1$ child nodes are valid trees, the number of trees we need to consider is bounded by

$$C_n m! = C_{m-1} m! = \frac{(2m-2)!}{(m-1)!} = \frac{(2n)!}{n!}. \tag{4.4}$$

Some values for this formula are given in Table 4.1. This sequence can be found in the *On-Line Encyclopedia of Integer Sequences* as sequence A001813 (Quadruple factorial numbers).

m	1	2	3	4	5	6	7	8	9
C_{m-1}	1	1	2	5	14	42	132	429	1 430
$C_{m-1}m!$	1	2	12	120	1 680	30 240	665 280	17 297 280	518 918 400

Table 4.1: The number of binary trees we need to consider for m leaves.

There are a number of different algorithms to compute the rank of a binary tree and its inverse operation, which can be used to enumerate all binary trees with a certain number of nodes [Kno77, RV78, Pro80, SF80, Zer85, RP90, Spr92, LvBR93]. The algorithm described in [SF80] is quite elegant and allows for fast computation of successive binary trees. However, schemes such as [Kno77] that rely on unranking allow for easier parallelization.

The free Mathematica demonstration applet Binary Tree Enumeration gives a nice graphical overview over the different possible binary trees. In our case, enumeration performance actually is not critical since all C_n binary trees can easily be precomputed for e.g. $n < 8$ with $C_7 = 132$. It is much more important to efficiently prune branches of the permutation tree for the leaves. Heap's permutation algorithm to generate all permutations is suited for this [Sed77, Lip79, Knu05]:

```
void generate(const int n) {
```

```
  if (n == 1) { // full permutation
    run();
    return;
  }

  if (!test(n)) // prune check
    return;

  // generate rest of permutation
  for (int i = 0; i < n - 1; i++) {
    generate(n - 1);
    swap((n & 1) ? 0 : i, n - 1);
  }

  generate(n - 1);
}
```

To prune a branch, we assume that the bounding boxes of all leaves that are not yet determined by the permutation prefix are empty. If the summed area then exceeds the currently best known value, no permutation in the permutation subtree will achieve a better value, therefore it can be pruned.

There are often two binary trees in the enumeration that are symmetric to each other, therefore the summed area is equal for both. One could optimize the evaluation algorithm if an easy way was found to identify these symmetric trees. Similarly, if a permutation where only sibling leaves are swapped does not make a difference.

Note that this tree-optimization algorithm can be applied multiple times. Table 4.2 lists the improvement of summed node area for some scenes after a single run, where we chose $m = 8$.

	Sponza	Conf. (simpl.)	Conf.	Kitchen (transf.)	Interior09	Buddha
Summed area before	154307	842865	8412.97	8281.78	398678	3.94301e8
Summed area after	139191	731707	7309.09	6843.93	344896	3.90666e8
Summed area reduction	9.77%	13.19%	13.12%	17.36%	13.49%	0.92%
Max. tree depth before	21	22	27	25	33	26
Max. tree depth after	28	26	34	34	41	32
Avg. leaf depth before	16.36	16.27	19.93	19.26	21.97	19.54
Avg. leaf depth after	18.48	18.78	24.13	23.78	26.00	19.88

Table 4.2: Improvements of summed node area, measured for the whole tree. Only one iteration of the optimization algorithm has been performed.

The reduction of summed node area is not as large on additional runs. We conjecture that the result converges to a tree where *every* subtree with at most m leaves is optimal in the sense of minimized summed node area. We found the frame rate improvement to be approximately proportional to the reduction of summed node area. Note that we did only test primary rays yet. It is interesting that while the summed area decreases, the average leaf depth actually

increases for all the scenes. Figure 4.1 shows the improvement by displaying the number of traversed nodes before and after the optimization.

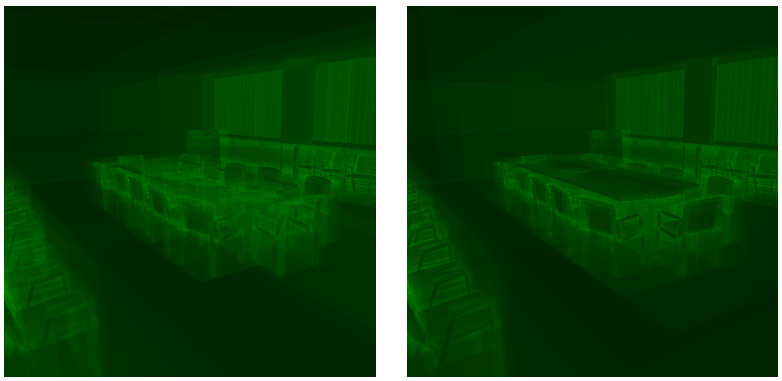


Figure 4.1: The number of traversed nodes before (left) and after (right) the optimization.

It is clear that the greatest reduction of overall summed area can be achieved on levels near the root, since the bounding boxes are largest there. It is therefore apparent to try to only optimize these levels, with the additional benefit of a small number of subtrees to process. We tried this setting for the first 10 levels from the top, which guarantees a number of subtrees smaller than $\sum_{i=0}^{9} 2^i = 2^{10} - 1 = 1023$. The results were not quite as good as optimizing the whole tree, but a considerable reduction of summed node area was achieved. Results for the massive "power plant"-scene can be found in Table 4.3. It is noteworthy that the rendering time increased very slightly after the first iteration despite the reduced summed node area. However, after another iteration, the optimized tree yielded superior results compared to the original tree.

	Before	Iter. 1	Iter. 2	Iter. 3	Iter. 4
Summed area	1.36267e13	1.27797e13	1.25772e13	1.25063e13	1.24589e13
Summed area reduction		6.22%	1.58%	0.56%	0.38%
Avg. leaf depth	27.57	33.29	33.05	33.03	33.12

Table 4.3: Results for running the optimization algorithm multiple times on the "power plant" scene. Only the ten lowest levels (near the root) have been considered and the largest subtrees considered had 8 leaves.

Due to the large amount of time it takes to optimize a tree, this method is only suited for static scenes, where the preprocessing time is amortized by shooting a large number of rays.

Further Possible Improvements

- If the SAH formula is interpreted as an upper bound for the summed area of the current subtree, it might be beneficial to improve this upper bound, i.e. make it sharper. We implemented one recursive split step, that is for each possible split position, we

"simulated" the split and computed the upper bound for the summed area of the corresponding subtree using the SAH formula. Unfortunately this sharper bound resulted in decreased performance, and the summed area of the whole tree increased.

- To decide upon a local split position (maybe using a binning approach), we build the underlying tree up to a certain level using a fast SAH binning approach for each possible split position. The tree with the smallest summed node area determines the split position.

- Only for a very small number of primitives we are able to generate trees that minimize the summed area of the bounding boxes of the nodes. One could therefore use the resulting tree of the standard SAH build algorithm as a guide tree. Apparently, the SAH clusters objects very efficiently, but since it is a greedy heuristic, this can be improved globally. This approach will be described in more detail in the next section.

4.1.3 Relations to R-Trees

In the field of spatial data management, the R-tree [Gut84] is a widely used data structure that allows for efficient indexing of rectangular objects. The basic concept behind R-trees is very similar to BVHs and kd-trees (in the case of the R*-tree [BKSS90]) and a lot of variants and improvements have been developed over the last decades in the database community. Nevertheless there is a very important difference to the typical acceleration structures used for ray tracing: While R-tree-like data structures allow for fast queries regarding *all* intersections with a given object (i.e. a range search), for ray tracing we usually only need the *first* hit along the ray, usually implemented using a backtracking search.

- The original R-tree paper [Gut84] describes a quadratic split algorithm. If the triangles of a mesh are numbered similar to a triangle fan, only one of the partitions grows in each step. Consequently, the tree degrades to something similar to a linear list. Similar problems have been observed for single-linkage clustering algorithms.

- The R*-tree [BKSS90] addresses this issue, but it does not favor splits that divide the primitives evenly along the partitions. For R-trees this is not necessary, since the height is logarithmic anyway, similar to B-trees.

- Space filling curves can be used as an alternative to sorting along three axes. Such curves provide a mapping from high-dimensional points to one-dimensional indices and have been used successfully in the context of R-trees [KF94]. Interestingly, the index values are often similar when the high-dimensional points were spatially located near each other. The resulting index-array can be used for splitting, without having to update this array after a split (as is the case when using three axes). However this performed worse than the SAH for the Hilbert curve and catastrophical for the z-order curve, which does not preserve spatial locality as well.

  While space filling curves are defined for integer coordinates (and corresponding values along the curve), this can be extended to floating point numbers due to their fractal

nature. Ready-to-use code for comparing multidimensional floating points according to their Hilbert value can be found at [Moooo].

- Similar to the cR-tree of [BPT02], the splitting algorithm to find two partitions can be seen as a clustering problem. Using the k-means algorithm for $k = 2$ and measuring the quality of the resulting partitions using the SAH, this performed slightly worse than using three axes. However one could try to use the average silhouette width cluster quality measure described in [BPT02] instead. This measure does not account for the number of cluster entries, so resulting trees might be very deep.

- Similar to kd-tree constructions, most BVH traversal algorithms select the order of traversing the child nodes depending on the ray direction [Wal07]. When using space filling curves or clustering algorithms, it is not clear how to set this axis for each inner node. One possibility is to choose the axis that provides the least overlap (respectively the most free space) between the partitions.

  However performance increased when we traversed the child node first whose bounding box got hit first by the ray, even for the original SAH split algorithm that used three axes. This traversal scheme was actually already proposed in [KK86]. As an additional bonus, two bits of each inner node can be used for something else.

  For R-tree constructions, this is not a problem since *all* intersections must be determined for most queries in the context of spatial databases. However, ray tracing kernels usually only need to find the first intersection along the ray (or the presence of any intersection at all for shadow rays). An exception to this might be the algorithm described in [DSDD07] or analytic motion blur algorithms [KB83]. In such cases a higher branch factor (as usually used with R-trees) might be interesting as well.

4.1.4 Relations to Clustering

It is obvious that clustering algorithms are similar in their nature to the construction of bounding volume hierarchies. Both approaches try to distribute a set of small primitives into distinct partitions. The k-means algorithm is one of the most used clustering algorithms. Given n points $\mathbf{x}_1, \ldots, \mathbf{x}_n$ it creates k partitions of these points (clusters) $C_1, \ldots, C_2$ with the attempt of minimizing

$$\sum_{i=1}^{k} \sum_{\mathbf{x}_j \in C_k} \left(\mathbf{x}_j - \mathbf{c}_i\right)^2, \tag{4.5}$$

where $\mathbf{c}_i$ denotes the centroid of the i-th cluster, given by

$$\mathbf{c}_i = \frac{1}{|C_i|} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j. \tag{4.6}$$

Implementations are typically based on Lloyd's relaxation technique [Llo82] or optimized variants thereof (e.g. by using nonuniform seeding [AV07]).

A Suitable Metric for AABBs

As described above, the k-means algorithm is only suited for point clouds. However [BPT02] used k-means clustering in the splitting step of R-trees. Their measure is defined as follows:

- The distance of two AABBs b_0, b_1 is given by the length of the diagonal of the AABB that encloses both b_0 and b_1.
- The mean of AABBs is given by their center of gravity. Brakatsoulas et al. used two-dimensional AABBs, so they used area as a weight. The intuitive generalization to three dimensions would be volume. However area is suited better even in 3D-space since "flat" AABBs have zero weight otherwise.

We have implemented a hierarchical 2-means (i.e. $k = 2$, also known as bisecting k-means) acceleration structure builder using the measure described above. We also tried treating the triangles as points by using the center of their AABB (which is a good approximation for scenes with many triangles about the same size). Unfortunately, the results were always inferior than trees built using the standard SAH.

Why k-Means is Ill-Suited for AABB Hierarchies

Given two objects in a setting similar to Figure 4.2, the result of 2-means as a clustering algorithm is perfect: Objects A and B are successfully assigned to different clusters. However the AABBs of both clusters are disastrous from the perspective of building acceleration structures: The AABB of object A is enclosed completely in the AABB of object B! The SAH heuristic would have split the object B in between and created two AABBs with a much reduced overlap. Given tighter object bounds used for the acceleration structure, this problem could be ameliorated. For example oriented bounding boxes (OBBs) could be used (these can be generated efficiently using principal component analysis [Gotoo]). However such bounds introduce overhead in the traversal and cause numerical instabilities compared to AABBs.

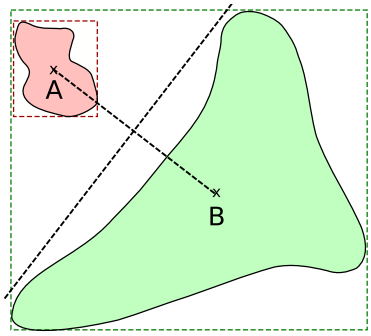


Figure 4.2: The k-means algorithm does not account for the overlapping of cluster AABBs. (We assume both objects A and B to be tesselated; this is not depicted here.)

Clearly this result is due to the L_2-norm used for measuring distances in the k-means algorithm. A suitable replacement for AABBs would be the L_∞-norm, i.e. $\|\mathbf{x}\|_\infty = \max_i x_i$ for $\mathbf{x} = (x_1, \ldots, x_n)$. But even then we quickly end up in a situation depicted in Figure 4.3.

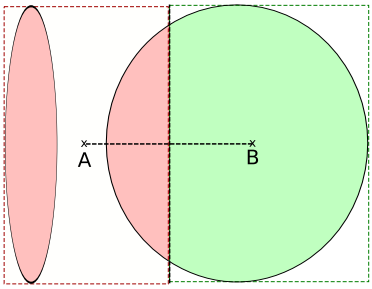


Figure 4.3: Using the L_∞-norm, the sum of the AABB surface areas not minimized. (Again, we assume both ellipsoids to be tesselated.)

This result is caused by an unfortunate placement of seeds. However even if the seeds were placed to assign each ellipsoids to a different cluster, we can easily construct scenes where one of the objects is extremely small compared to the other cluster. This is undesirable as most rays will not hit the small object. Therefore the majority of rays does not benefit from such a split. Instead we need to consider the tradeoff between small overlap and the relative bounding box sizes. We conclude another difference to the SAH heuristic: k-means does not attempt to create about equally sized clusters and it completely disregards the number of primitives per cluster. Yet both factors are very important for acceleration structures. Since the number of primitives per child is an indicator of the remaining tree depth it only makes sense to split clusters with few primitives when their AABB size is large (and consequently many ray queries will benefit from this split).

If we try to incorporate these demands we arrive at something very similar to the existing SAH heuristic: We argued that the L_∞-norm is better suited when AABB hierarchies are generated. In order to respect all partitions (to avoid the result of Figure 4.3) we need to "scan" along all cardinal axes. Finally we want to favor partitions that are favorable for the majority of rays. Therefore we need to account for the size of the cluster AABBs and the number of primitives per cluster. Both of these factors appear in the SAH heuristic (see Equation 4.1).

Cluster Quality Measures

It is known that Lloyd's implementation of the k-means algorithm tends to get stuck in local minima. Therefore in practice the clustering step is repeated a number of times. To find the best clustering result a quality measure is needed. Of course the SAH cost function can be used for this but as described above, the results are likely to be very similar to the usual SAH build.

The *average silhouette width* is a classic clustering quality measure that was used in [BPT02] as well to determine the number of clusters k for the k-means algorithm. However to determine the silhouette width of a cluster many nearest neighbors have to be found. When implemented naively the runtime for the quality measure alone is $\mathcal{O}\left(n^2\right)$, rendering it impractical for realistic scence sizes. Building a kd-tree to accelerate the nearest neighbor queries would certainly help. But building a spatial hierarchy in order to build an acceleration structure is inefficient as well. Additionally, the average silhouette width does neither respect cluster

AABB overlapping or the number of cluster primitives since it is based on distances to nearest neighbors only.

Nevertheless, given a more suitable hierarchical clustering approach that produces more than two clusters, a good binary tree could be generated using the approach of Subsection 4.1.2.

Building optimized acceleration structures for ray tracing is an ongoing area of research. It recently gained a lot of attention with the advent of interactive and real-time ray tracers that can handle dynamic scenes. In contrast to previous offline rendering systems not only the quality of the acceleration structure plays a key role with these interactive systems but also the time taken to build the structure. An excellent survey of the different approaches developed in the last few years can be found in [WMG+07].

For offline rendering systems, the motion paths of objects are known completely in advance. It is clear that such systems can exploit this information to speed up the rendering process. In contrast online systems such as games are much more challenging: The motion of objects in the scene depends on the unpredictable decisions of the players.

4.2 Dynamic Acceleration Structures

While [WMG+07] focus on handling dynamic scenes in a ray tracing context, they do not tackle the problem of simulating motion blur. Instead by "dynamic" they mean that geometry may change *in between* frames but stays statically during the shutter interval of a single frame. Nevertheless a lot of the results presented by Wald et al. can directly be transferred to a setting where motion blur simulation is desired:

- Motion blur is especially important for animation rendering. All the approaches that exploit coherence between frames still apply for the whole animation. Instead of tight bounding boxes for the static objects per frame we can use bounding boxes that enclose the object during the whole shutter interval.

  If the motion is constrained to linear segments, the bounding boxes can often be fit more tightly by storing the bounding boxes for the start and the end of the shutter interval at each node. For a single point in time, the linearly interpolated bounding box often provides a rather tight bound. This approach is used in Pixar's PhotoRealistic RenderMan [CFLB06].

  Unfortunately Christensen et al. do not describe over which bounding boxes they build their acceleration structure. One obvious possibility is taking the bounding boxes for the full shutter interval $T = [t_0, t_1)$ to build the tree and later interpolate the bounding boxes in the traversal for each ray. Depending on the kind of motion it might be better to build the tree for a time instant, e.g. $\frac{t_0+t_1}{2}$ in anticipation that a good hierarchy looks similar for the rest of the shutter interval. We would like to note that the memory requirements of each node will rise, since two bounding boxes need to be stored. For fast movements, this will often pay off nevertheless, due to the much tighter bounding boxes.

- Usually super-sampling is absolutely necessary in a production environment to yield high-quality anti-aliasing. Consequently we can subdivide the shutter interval T to

achieve well-stratified temporal sampling points. Each such subinterval $T_1, \ldots, T_n$, with $T_i \cap T_j = \varnothing$ for $i \neq j$ and $\bigcup_{i=1}^{n} T_i = T$, can be interpreted to belong to a different frame $F_1, \ldots, F_n$ we want to render. The final frame F then corresponds to a convex combination of these "sub-frames" weighted by the relative length of the shutter subinterval:

$$F = \sum_{i=1}^{n} \frac{|T_i|}{|T|} F_i. \tag{4.7}$$

Instead of building a single acceleration structure to render all the subframes F_i, we can build a corresponding hierarchy for each F_i. Of couse the time saved by using such more "fitting" acceleration structures should exceed the additional cost for building this multitude of structures for this approach to be effective.

There are three major categories on how to handle changed geometry settings in a scene, or more precisely how to proceed when the acceleration structure tree is invalidated. We shortly describe those in the following sections and refer the reader to [WMG+07] for a more detailed treatment.

4.2.1 Rebuild from Scratch

This is arguably the most robust and simple strategy. The old tree is discarded and a new tree is built using the current geometry, possibly in a lazy fashion. Fast tree building heuristics such as [WK06] are especially important then. This method is very robust since it implies no restrictions on the geometry and movement at all – even the topology may change arbitrarily.

4.2.2 Incremental Updates

The idea here is to exploit the coherence typcially given is subsequent frames (or subintervals of the shutter interval, as described above). When the movement is not arbitrarily and the topology stays the same, often whole subtrees of the acceleration structure can be reused. Therefore the approach of incremental updates tries to reuse as much information as possible from the previous acceleration structure to reduce the amount of work to yield an updated tree.

While these updates usually work quite well for a few subsequent steps, the acceleration structure tends to degrade afterwards. Therefore hybrid schemes have been proposed that combine (possibly asynchronously) full rebuilds and updates of the acceleration structure, using different quality measures to guide selective restructuring [YCM07].

In general BVHs are much more suited for this kind of incremental update. That is because the hierarchy itself can be kept while only the primitive bounding boxes need to be updated and propagated bottom-up (known as "refitting"). This works in $\mathcal{O}(n)$ for n primitives, however the resulting tree quality very much depends on the kind of motion present in the scene. It works especially well when the hierarchy of the tree would stay very similar if the tree was rebuilt from scratch.

4.2.3 Multiple Hierarchies

Instead of using a single hierarchy for the whole scene one can recognize the parts with moving geometry and keep a separate acceleration structure for the static parts [HDMS03]. This only works for some settings, e.g. a moving character in a static environment. If the movement of the character is repetitive it may pay off as well to precompute matching acceleration structures for each pose and reuse them.

Usually such a strategy is especially useful for two-level hierarchies: Instances that reference other acceleration hierarchies are stored together with an transformation matrix. Whenever the instances are moved, only the transformation matrices need to be changed. The traversal then simply applies the inverse transformation before recursively traversing the lower-level acceleration hierarchies.

4.3 Static Acceleration Structures

Instead of updating the acceleration structure one can build one acceleration structure for the whole animation. In this context "animation" means over an extended period of time, either for a single frame with moving objects or multiple frames.

4.3.1 Splitting the Time Dimension

As described above one can simple enlarge the bounding boxes of moving objects to enclose the object during the whole time interval and build an acceleration structure over these enlarged bounding boxes. It is clear that this approach may produce very bad hierarchies for some scenes. A pathological case is a small triangle that moves along the diagonal of the bounding box of the whole scene during the shutter interval, since its bounding box then is as large as the whole scene. While the subdivison of the shutter interval helps to ameliorate this, the basic problem remains.

For the following approaches it is helpful to recognize moving 3D-geometry as static geometry in 4D-space. Therefore it seems logical to build an acceleration structure by subdividing this 4D-space.

[Gla88] describe a four-dimensional structure that can be characterized as a hybrid of adaptive space subdivision and bounding volume techniques. Its goal is to combine the good hierarchies of space subdivision with the tight bounds of bounding volumes. First a top-down octree-like space-subdivision is run on the objects of the scene (or more precisely, a "hex-tree" since four dimensions are used for subdivison), based on some split-criterion. Afterwards, a bottom-up process finds non-overlapping bounding volumes for the objects of each cell. Obviously, this results in object duplication, similar to kd-trees. The regular subdivision steps resembles some similarity to the global heuristic proposed in [WK06].

A similar subdivision for the time-axis is applicable for kd-trees as well. However it is not clear when to select the time-axis for a split and where to put the plane for those splits. For example, the surface area heuristic (SAH) [GS87, Havoo] cannot be easily extended since a split in time does not partition the object list, but can only reduce the space filled by the

geometry over time. In addition, there are no candidate splitting planes available in time, whereas for splitting in space it suffices to consider planes at the vertices of triangles.

To deal with these problems, [Ols07] propose the following extension:

- Candidate split positions are found in a pre-processing step: For each primitive P, the shutter interval is partitioned, such that for each partition, the corresponding enclosing 3D bounding volume area does not exceed $\frac{1}{\kappa}B$, where B denotes the area of the 3D bounding volume corresponding to the whole shutter interval. The value κ therefore restricts how much the animated spatial bound is allowed to grow. Olsson proposes a value of $\kappa = 2$, but this parameter can be chosen per primitive as well.

  New primitives $\tilde{P}_i$ are generated that correspond to the the original primitive P, but are restricted to the movement of the i-th shutter interval partition. To summarize, each primitive P is subdivided over time to generate new primitives $\tilde{P}_i$ whose bounding volume areas are constrained.

- The splitting step of the kd-tree construction then works on these subdivided primitives. The partitions of the shutter interval are considered as candidate splitting planes over time. For splits along the time axis, the conditional probability of the SAH is modified to equal $\frac{|T_L|}{|T|}$ and $\frac{|T_R|}{|T|}$, where T_L and T_R denote the time intervals corresponding to the child nodes and T denotes the assigned time interval of the node that needs to be split. The primitive counts of the SAH formula then apply to the subdivided primitives $\tilde{P}_i$. In the leaf, a reference to each original primitive P suffices since the motion path of the subdivided primitives P_i equal that of P for their respective time intervals.

The traversal code must check for these splits along time as well. However no division is necessary for this case, in contrast to splits in space. Instead only the correct child must be chosen for traversal depending into which child node time interval the ray's time sample falls.

Olsson notes that the memory requirements are quite high for his approach since the primitive subdivision can lead to an enormous increase of the primitive count. This also increases the complexity of the kd-tree build significantly because of the required sorting along the axes.

Subdividing primitives over time as a preprocessing step to reduce the size of the bounding boxes is possible for BVHs as well. Since usually the objects are visible during the whole frame, partitioning object lists by splitting the time dimension does not make sense. Instead new objects have to be added "artificially" by subdividing the primitives along time. To preserve the $\mathcal{O}(n)$ memory complexity property of BVHs, the number of subdivisions for each primitive may not exceed a constant.

4.3.2 Ray Classification

The ray classification approach by [AK87] has been extended by [GP90a] and [Qua96] to handle time-continuous animations. Arvo et al. classify rays using their origin 5D-points and collect them into 5D-hypercubes. A 5D-hypercube can also be considered as a beam in 3D-space. For each such beam, a candidate set of primitives is found whose bounding

volume overlaps the ray. When a ray is shot, the primitives its corresponding candidate set are intersected. Instead of subdividing 5D-space evenly, a hierarchy of 5D-hypercubes can be built as well, using binary subdivision on demand where hypercubes are too large.

This principle can easily be extended to 6D-bounding volumes when time is considered an additional dimension. The binary subdivision step now occurs along six axes compared to the previous five and the beams need to be tested against 4D-bounding volumes. This corresponds to intersecting a 3D-beam against the bounding volume of the primitive for the time interval of the 6D-hypercube.

Unfortunately it is rather expensive to classify whether an object is in a beam, therefore ray classification methods are currently believed to yield inferior performance compared to BVHs and kd-trees.

Ray classification is the only currently known algorithm that transforms the typical backtracking search of acceleration structures to a pure search problem. Of course the intersection results of the candidate primitive set still have to be processed to find the first intersection, but both locating this minimum and finding the candidate set works without backtracking.

5 (t, m, s)-Nets and (t, s)-Sequences

The Monte Carlo method uses purely random samples. In this chapter we describe a certain kind of *low-discrepancy* sampling construction that replaces these random samples by completely deterministic ones. In practice the creation of truly random bits is quite slow, so we have to rely on deterministic pseudo-random numbers anyway. Yet the choice of the pseudo-random number generator is crucial for fast convergence.

5.1 Higher Uniformity of Quasi-Monte Carlo Point Sets

Low-discrepancy sampling contructions attempt to yield points "as uniform as possible". In contrast to random points the samples of a low-discrepancy point set are *not independent* due to their deterministic nature. Using such point sets in multivariate integration is referred to as quasi-Monte Carlo integration.

5.1.1 Discrepancy

To determine uniformity, we can use the the *star-discrepancy* of a point set which measures its distance between the empirical distribution induced by the point set and the uniform distribution [Lemo8]. Consider the set of rectangular boxes with a center at the origin

$$B(\mathbf{u}) = \left\{(x_1, \ldots, x_s) \in [0,1)^s : \ 0 \le x_j \le u_j, \ 1 \le j \le s\right\}, \tag{5.1}$$

where $\mathbf{u} = (u_1, \ldots, u_s) \in [0,1)^s$. For a point set $P_n = \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$ we then count how many samples $\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,s})$ fall into such a box, denoted by

$$\alpha(P_n, \mathbf{u}) = \left|\left\{\mathbf{x}_i : \ 0 \le x_{i,j} \le u_j, \ i = 0, \ldots, n-1\right\}\right|. \tag{5.2}$$

The star-discrepancy $D^*(P_n)$ of a point set now compares the departure of P_n from uniformity by comparing the empirical probability $\alpha(P_n, \mathbf{u})/n$ to the value $\prod_{j=1}^{s} u_j$ corresponding to uniform distribution over $[0,1)^s$ using the Kolmogorov-Smirnow statistic:

$$D^*(P_n) = \sup_{\mathbf{u} \in [0,1)^s} \left| \prod_{j=1}^{s} u_j - \frac{\alpha(P_n, \mathbf{u})}{n} \right|. \tag{5.3}$$

Low-discrepancy constructions try to yield small star-discrepancy values by avoiding the clumping (or clustering) that is typical for random point sets. While there are other discrepancy measures, the importance of the star-discrepancy stems from the Koksma-Hlawka theorem which derives a deterministic upper bound on the integration error:

$$\left| \frac{1}{n} \sum_{i=1}^{n} f(\mathbf{x}_i) - \int_{[0,1)^s} f(\mathbf{x}) d\mathbf{x} \right| \le D^*(P_n) V(f), \tag{5.4}$$

where $V(f)$ denotes the *variation in the sense of Hardy and Krause* of the function f that is integrated. Unforunately $V(f)$ is infinite for all non-trivial functions in computer graphics due to the ubiquity of discontinous functions (e.g. triangle borders).

While this error bound is not valid in the domain of computer graphics, this does not imply that using quasi-Monte Carlo samples is not beneficial. The contrary is true – the rate of convergence increases significantly in empirical experiments [KK02b]. Yet in practice care must be taken to hide the inherent correlation of quasi-Monte Carlo samples. The discrepancy measure has been used first in computer graphics by [Shi91].

5.1.2 Minimum Distance

Another quality measure for point sets is minimum distance, defined as

$$d_{\min}(P_n) := \min_{0 \le i < j < n} \left\| \mathbf{x}_i - \mathbf{x}_j \right\|, \tag{5.5}$$

where $\|\cdot\|$ usually denotes the Euclidean L_2-norm. However when the point sets are meant to be used as tiles, the toroidal distance might be more appropriate [GHSK07]. The toroidal measure is also the right choice when a Cranley-Patterson is applied to the point set – the minimum distance of the point set stays the same then.

Similar to small discrepancy, point sets with a large minimum distance seem to improve the rate of convergence. The poisson-disk dart throwing algorithm [WCE07] or Lloyd relaxation [HDK01] are classic methods to generate such sets with a blue noise spectrum. Such point sets have been extensively used in computer graphics [Coo86, HA90, Shi91, PH04]. However rank-1 lattices (see [Lem08, Sec. 4.3] for an introduction to lattices) and (t, m, s)-nets (which will be described below) have recently been optimized for maximized minimum distance as well [DK07, GHSK07]. Maximizing minimum distance is a fundamental concept that appears regulary in nature [Yel83] [DL04, Ch. 8].

5.1.3 Dispersion

Yet another quality measure for point sets is the *dispersion* $d_{\text{disp}}(P_n)$ (also known as the *covering radius*) [YLVA00]. Let

$$B(\mathbf{c}, r) = \left\{ \mathbf{x} \in [0,1)^s : \ \|\mathbf{x} - \mathbf{c}\| \le r \right\} \tag{5.6}$$

denote the closed ball of radius r centered at $\mathbf{c}$. The dispersion of a point set P_n is the minimal radius r such that the balls $B(\mathbf{x}_0, r), \ldots, B(\mathbf{x}_{n-1}, r)$ cover $[0,1)^s$, i.e.

$$d_{\text{disp}}(P_n) = \sup_{\mathbf{x} \in [0,1)^s} \min_{0 \le i < n} \left\| \mathbf{x} - \mathbf{x}_i \right\|. \tag{5.7}$$

Again, the norm $\|\cdot\|$ can be chosen appropriately. The smaller the dispersion, the better the point set is.

5.1.4 Extensive Stratification

One way to generate well-stratified samples for quasi-Monte Carlo integration is the use of (t, m, s)-nets and (t, s)-sequences. We will give definitions for both shortly. A much more sophisticated treatment of the topic can be found in [Nie92, Ch. 4].

For the subsequent definitions, we fix the dimension $s \geq 1$ and an integer $b \geq 2$.

Definition 3 ([Nie92, p. 48]). A subinterval E of $[0,1)^s$ of the form

$$E = \prod_{i=1}^{s} \left[a_i b^{-d_i}, (a_i + 1) b^{-d_i} \right), \tag{5.8}$$

with $a_i, d_i \in \mathbb{Z}$, $d_i \geq 0$, $0 \leq a_i < b^{d_i}$ for $1 \leq i \leq s$ is called an *elementary interval in base b*.

See Figure 5.1 for a two-dimensional illustration.

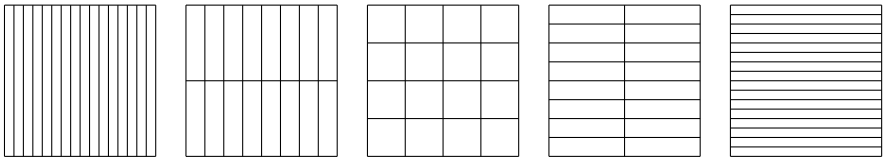


Figure 5.1: All kinds of elementary intervals with area $\frac{1}{16}$ for $s = b = 2$.

Definition 4 ([Nie92, Def. 4.1]). For integers $0 \leq t \leq m$, a (t, m, s)-*net in base b* is a point set of b^m points in $[0,1)^s$ such that there are exactly b^t points in each elementary interval E with volume b^{t-m}.

The parameter t controls the quality of the net. We are especially interested in nets where there is exactly one point in each elementary interval, i.e. $t = 0$. An example of such a net is given in Figure 5.2.

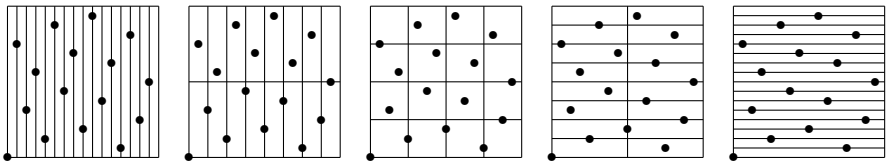


Figure 5.2: A $(0, 4, 2)$-net in base 2. There is exactly one point in each elementary interval.

Note that for $(0, m, s)$-nets there is exactly one point in each column and in each row (the kind of elementary intervals at the very left and the very right in Figure 5.2). This is known as Latin Hypercube stratification. But moreover, there is also exactly one point in each square elementary interval (the kind of elementary intervals in the middle). Such a stratification is usually desired when jittered-grid sampling is used. The concept of $(0, m, s)$-nets not only combines both properties, but even extends them (because the other kinds of elementary intervals are neither covered by Latin Hypercube nor jittered-grid sampling).

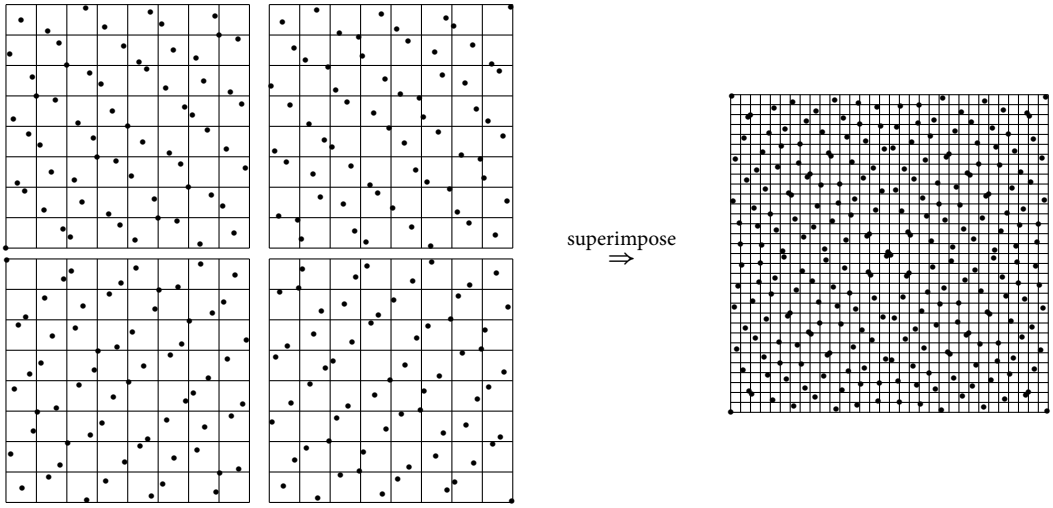


Figure 5.3: Each point set $\mathbf{x}_{k \cdot 2^6}, \ldots, \mathbf{x}_{(k+1)2^6-1}$ of the Sobol' $(0, 2)$-sequence is a $(0, 6, 2)$-net in base 2 for $k \geq 0$. Depicted here are the nets for $k = 0, 1, 2, 3$. When these nets are superimposed, they form a $(0, 8, 2)$-net in base 2, shown on the right.

Definition 5 ([Nie92, Def. 4.2]). For an integer $t \geq 0$, a sequence $\mathbf{x}_0, \mathbf{x}_1, \ldots$ of points in $[0, 1)^s$ is a (t, s)-*sequence in base* b if, for all integers $k \geq 0$ and $m > t$, the point set $\mathbf{x}_{kb^m}, \ldots, \mathbf{x}_{(k+1)b^m-1}$ is a (t, m, s)-net in base b.

Figure 5.3 illustrates the concept of (t, s)-sequences. There are a number of constructions for (t, s)-sequences, including

- Sobol' sequences in base 2 [Sob67], covered in more detail in the next section,
- Faure sequences [Fau82], which can be seen as a generalization of Sobol's sequence for prime bases,
- generalized Faure sequences [Nie87] for prime power bases,
- Niederreiter sequences [Nie88] (all sequences above are special cases of Niederreiter sequences)
- Niederreiter-Xing sequences [NX96a, NX96b].

5.2 Matrix-Generated (t, m, s)-Nets and (t, s)-Sequences

Let q be a prime power and let F_q be the finite field with q elements. Given suitable $m \times m$-matrices $C_1, \ldots, C_s$ over F_q, the j-th component of the i-th point for $1 \leq j \leq s$ and $0 \leq i < q^m$ can be generated by

$$x_i^{(j)} = \begin{pmatrix} b^{-1} \\ \vdots \\ b^{-m} \end{pmatrix}^T \left[C_j \begin{pmatrix} d_0(i) \\ \vdots \\ d_{m-1}(i) \end{pmatrix} \right] \in [0, 1), \tag{5.9}$$

where the matrix-vector multiplication in brackets has to be performed in F_q and $d_k(i)$ represents the q-ary expansion of i:

$$i = \sum_{k=0}^{m-1} d_k(i) q^k. \tag{5.10}$$

Most constructions of (t, m, s)-nets describe the creation of suitable matrices $C_1, \ldots, C_s$. A criterion of when such generator matrices are suited to generate $(0, m, s)$-nets will be given in Theorem 1. A more general formulation of this construction principle can be found in [Nie92, Sec. 4.3].

(t, s)-sequences can be constructed in a very similar way to Equation 5.9. However, since the number of points is unlimited in a sequence, the size of the generator matrices is infinite. This works in practice, since $d_k(i) = 0$ for all sufficiently large k and in addition, we require the matrix entries $c_{pr}^{(i)} = 0$ for all sufficiently large p [Nie92, p. 72, (S6)]. Therefore only finite upper left submatrices have to be considered to generate points.

The van der Corput sequences are a famous example of $(0, 1)$-sequences. For them, the generator matrix is equal to the infinite identity matrix.

5.2.1 The Sobol' Sequence

For global illumination rendering samples that are high-dimensional and well-stratified are very desirable. This need even increases if features like anti-aliasing (both spatial and temporal), depth of field, participating media or wavelength-dependent effects should be supported. The Sobol' sequence is ideally suited for this purpose. Due to its strictly deterministic nature, algorithm parallelization gets trivial by using different sequence starting indices for each process or thread.

The Sobol' sequence [Sob67] is the first construction in the family of digital nets [Nie92]. It is designed to work in base $b = 2$. We need to introduce some definitions for polynomials over finite fields before we describe the construction of the generator matrices. We refer the reader to [LN86] for an excellent introduction to finite fields. As an alternative to the Sobol' sequence, the Richtmyer sequence seems to be suited for high-dimensional quasi-Monte Carlo integration as well and is easier to create [TVC07].

Definition 6. A *polynomial* over a ring R is an expression of the form

$$f(x) = \sum_{i=0}^{n} a_i x^{n-i}, \tag{5.11}$$

where n is a nonnegative integer, the coefficients a_i are elements of R and x is a symbol not belonging to R, called an indeterminate over R.

Definition 7. The *degree* of such a polynomial, denoted $\deg(f)$ is $\max\{i : a_{n-i} \neq 0\}$. It is set to ∞ if all a_i are zero.

Definition 8. The polynomials over R with the operations of addition and multiplication form a *polynomial ring* over R, denoted $R[x]$.

Definition 9. Let $\mathbb{F}_q$ denote a finite field with q elements. For a nonzero polynomial $f \in \mathbb{F}_q[x]$ the least positive integer e for which $f(x)$ divides $x^e - 1$ is called the *order* of f, denoted by $\operatorname{ord}(f)$.

Definition 10. A polynomial $f \in \mathbb{F}_q[x]$ with $\deg(f) = s$ and $f(x) = \sum_{i=0}^{s} a_i x^{s-i}$ is a *primitive* polynomial if and only if $a_0 = 1$, $a_s \neq 0$ and $\operatorname{ord}(f) = q^s - 1$.

If the polynomial $f \in \mathbb{F}_q[x]$ is primitive, the smallest integer r for which $x^r \equiv 1 \pmod{f(x)}$ is $r = q^k - 1$. Consequently, the powers of x (modulo $f(x)$) from 0 to $q^k - 1$ generate the set of nonzero polynomials over F_q of degree less than k [LN86, Lemo8].

We now describe how the generator matrix of one component of the Sobol' sequence is constructed [BF88, PVTF02, JK03, Lemo8]. First we need an arbitrary primitive polynomial $f \in \mathbb{F}_2[x]$. If $s = \deg(f)$ and

$$f(x) = \sum_{i=0}^{s} a_i x^{s-i} \tag{5.12}$$

it follows that that $a_0 = a_s = 1$ since f is primitive.

Next, we need to choose m_k for $k = 1, \ldots, s$. These values can be chosen freely provided that each m_k is odd and less than 2^k. Based on the values $m_1, \ldots, m_s$ new values m_k for $k > s$ are obtained using the following recurrence:

$$m_k = \left(\bigoplus_{i=1}^{s} 2^i a_i m_{k-i} \right) \oplus m_{k-s}, \tag{5.13}$$

where $\oplus$ denotes an exclusive-or operations that corresponds to addition in $\mathbb{F}_2$. Then the k-th column of the generator matrix corresponds to the bits after the binary point in the expansion of $v_k = 2^{-k} m_k$. One can show that the resulting matrix is regular and has upper triangular shape. Consequently the matrix generates a $(0, 1)$-sequence in base 2 by Theorem 1.

The numbers v_k are called "direction numbers" since depending on the bits of the sample index, these columns of the matrix ("directions") are added together to form the sample.

To generate sequences with multiple components, a different primitive polynomial must be chosen for each component. To achieve a small quality parameter t, the degrees of these polynomials should be as small as possible [Lemo8]. By choosing the initial m_k used for the direction numbers carefully, additional stratification properties can be achieved. The SamplePack implementation by T. Kollig and A. Keller uses initial $m_k = 1$ for each component. Better results (the so called "Property A") can be achieved for up to 1111 dimensions using the freely available table of primitive polynomials and m_k values used in [JK03]. For the first component usually $m_k = 1$ is used for all k, yielding the van der Corput radical inverse. The use of good direction numbers is actually visible in images rendered using the Sobol' sequence, as shown in Figure 5.4.

Implementation

Unfortunately, previously available implementations are either complex [BF88, JK03] or only suited for low-dimensional points (typically up to 40 dimensions, such as the implementation

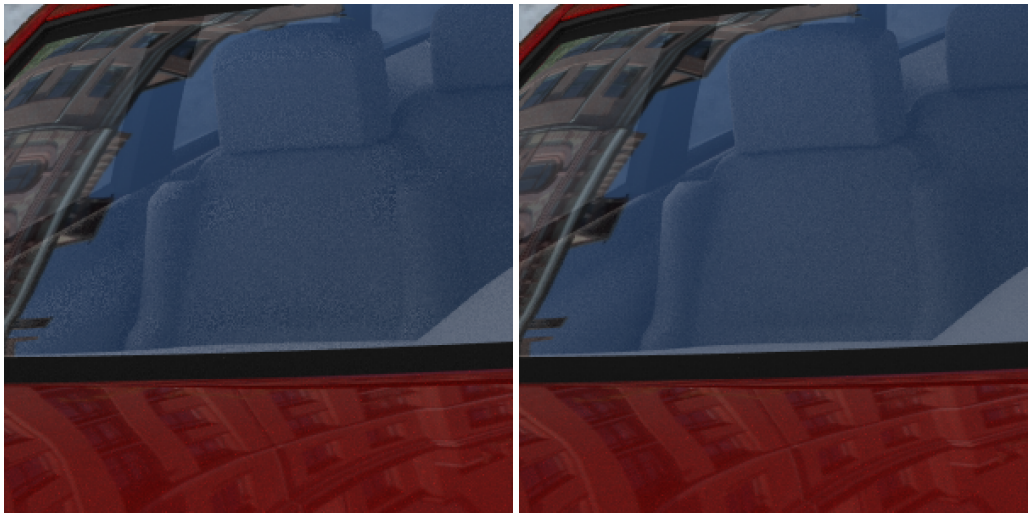


Figure 5.4: A close-up of a car seat behind a glass pane. On the left, Sobol' points were used with generator matrices created with initial $m_k = 1$ for each component. On the right, the initial values of [JK03] were used, yielding much less artifacts because of the better high-dimensional stratification.

in the GNU Scientific Library or the Numerical Recipes [PVTF02]). The commercial package SobSeq1024 is able to generate 1024-dimensional samples, but is closed-source.

The following code implements the construction of a generator matrix. The function expects a primitive polynomial $f \in \mathbb{F}_2$ (whose coefficients are encoded into the integer a) and the initial m_k values (for $k = 1, \ldots, \deg(f)$). Both theses inputs are available from the table mentioned above [JK03]. The function yields a matrix where each column is represented by an integer and the most significant bit corresponds to the first row of the matrix. The memory for the matrix is used to generate the remaining direction numbers, these are afterwards converted to the matrix columns.

```
int sobolGeneratorMatrix(const unsigned int a, const unsigned int * const m,
unsigned int * const matrix, const unsigned int matrixSize) {
  // determine degree of polynomial (could be optimized using __builtin_clz)
  int s = 0;
  for (int p = a >> 1; p; p >>= 1, s++);

  // first columns correspond to m_1,...,m_s
  memcpy(matrix, m, s * sizeof(unsigned int));

  // the remaining direction numbers are obtained by recurrence
  for (int k = s; k < matrixSize; k++) {
    matrix[k] = (matrix[k - s] << s) ^ matrix[k - s];

    // iterate over bits of polynomial
    for (int i = s - 1, p = a >> 1; i > 0; i--, p >>= 1)
      if (p & 1)
```

```
      matrix[k] ^= (matrix[k - i] << i);
  }

  // k-th column is the binary expansion of m_k / 2^k
  for (int k = 0; k < matrixSize; k++)
    matrix[k] <<= matrixSize - k - 1;

  return s;
}
```

Given the generator matrices, the computation of samples is both compact and highly efficient (and not limited to the Sobol' sequence), as the implementation below demonstrates. We would like to note that the number of dimensions should be specified before, although only as much components as needed are generated. However there is some overhead if not all components are used.

The implementation is not limited to the Sobol' sequence. Instead it can use any suitable generator matrices, e.g. the Niederreiter-Xing sequence is very well distributed for many dimensions. Unfortunately the construction of these generator matrices is extremely involved [NX96a, NX96b, Pir02].

The implementation assumes a special data layout of the generator matrices: Each column of the $\mathbb{F}_2^{32\times 32}$ matrices is stored as a 32-bit integer (with the most significant bit corresponding to the first row). Furthermore, the first columns of all matrices are stored sequentially, followed by all second columns and so on.

Instead of a full matrix-multiplication for each sample, we use an optimization that is valid when the samples are generated sequentially. Using Gray-code numbering [PVTF02], the order of each 2^m points is changed, but they are still stratified in the same way. By using Gray codes only one bit of the index changes per sample. Consequently, we only need to alter the previous result using the corresponding column of the generator matrix. In $\mathbb{F}_2$ this can be implemented efficiently using a XOR-operation. The drawback of using Gray code numbering is the limitation to sequential samples – it is not well suited to generate samples with arbitrary indices. Fast code to generate the samples corresponding to arbitrary indices (although inherently slower than using Gray codes) can be found in [Wäc08, Sec. 3.3.2].

The code below can make use of the SSE2 extensions of current processors and use optimized Intel intrinsics to find the changed bit of the Gray code (also see [War02] for other implementations of the `ntz` (number of trailing zeros) function). We are able to generate 2^{32} samples in 70 seconds on a single core of a 2.4 GHz Intel Quad Core processor.

```
class Sobol {
  unsigned int *matrices;
  unsigned int nDimensions;

  unsigned int *state;
  unsigned int index;

  // matrix initialization and cleanup omitted

  void setIndex(unsigned int pointIndex) {
```

```
    index = pointIndex;
    pointIndex ^= pointIndex >> 1; // Gray code representation

    memset(state, 0, sizeof(unsigned int) * nDimensions);

    const unsigned int *matrix = matrices;
    for (; pointIndex; pointIndex >>= 1, matrix += nDimensions)
      if (pointIndex & 1)
        for (unsigned int d = 0; d < nDimensions; d++)
          state[d] ^= matrix[d];
  }

  // update state for next point
  inline void next() {
    if (!++index) { // check for overflow
      setIndex(0);
      return;
    }

    // find the bit that has changed (Gray code)
#ifdef __GNUC__
    const int pos = __builtin_ctz((int) index);
#elif __INTEL_COMPILER
    const int pos = _bit_scan_forward((int) index);
#else
    int pos = 0;
    for (int mask = 1; !(index & mask); ++pos, mask <<= 1);
#endif

    // update the state
#ifdef __SSE2__
    __m128i *m = (__m128i *) (matrices + pos * nDimensions);
    __m128i *s = (__m128i *) state;
    const unsigned int maxD = nDimensions >> 2;
    for (unsigned int d = 0; d < maxD; d++)
      s[d] = _mm_xor_si128(s[d], m[d]);
#else
    unsigned int *m = matrices + pos * nDimensions;
    for (unsigned int d = 0; d < nDimensions; d++)
      state[d] ^= m[d];
#endif
  }

  inline double getComponent(const unsigned int c) const {
    static const double scale = 1. / (1ULL << 32);
    return state[c] * scale;
  }
};
```

5.2.2 Constructing $(0, m, s+1)$-Nets from $(0, s)$-Sequences

Every $(0, s)$-sequence can be use to generate a $(0, m, s+1)$-net for any $m > 0$ by adding the component ib^{-m} for the i-th point [Nie92, Lemma 4.22, p. 62]. The Larcher-Pillichshammer points [LP01] are widely used in computer graphics because of their good stratification properties [KK02b, PH04, DK07]. In the following we develop a method to modify constructions for $(0, s)$-sequences in order to integrate the Larcher-Pillichshammer points as a two-dimensional projection. The resulting nets are likely to perform superior in computer graphics applications.

Let $C_1, \ldots, C_s$ be infinite matrices over F_q. For any $m > 0$ we denote the upper-left $m \times m$-submatrix of a matrix M by $M(m)$.

Theorem 1 (see [Nie92, Thm. 4.36, p. 73])**.** Suppose that for any $m > 0$ and nonnegative integers $d_1, \ldots, d_s$ with $\sum_{i=1}^{s} d_i = m$, the system of the first d_1 rows of $C_1(m)$, together with the first d_2 rows of $C_2(m), \ldots$, together with the first d_s rows of $C_s(m)$ is linearly independent over F_q. Then the matrices $C_1, \ldots, C_s$ generate a $(0, s)$-sequence in base q.

Theorem 2. Suppose that the infinite matrices $C_1, \ldots, C_s$ over F_q fulfill the condition of Theorem 1 and therefore generate a $(0, s)$-sequence in base q. Further suppose that the infinite matrix C_1' over F_q also fulfills the requirements of Theorem 1 and therefore generates a $(0, 1)$-sequence in base q. Then a $(0, s)$-sequence in base q is generated by $C_1', C_2D, \ldots, C_sD$, where $D := C_1^{-1}C_1'$.

Proof. For any $m > 0$ and nonnegative integers $d_1, \ldots, d_s$ with $\sum_{i=1}^{s} d_i = m$, the system of the first d_1 rows of $C_1(m)$, together with the first d_2 rows of $C_2(m)$, ..., together with the first d_s rows of $C_s(m)$ is linearly independent over F_q. If we combine these rows in arbitrary order, we get an $m \times m$-matrix M that is regular. The matrix D is regular as well, since both C_1 and C_1' must be regular. Consequently, $M(D(m)) \in \mathrm{GL}(m)$ is regular and thus is a linearly independent system over F_q.

However, the rows of $M(D(m))$ are exactly the same as multiplying $C_1, \ldots, C_s$ with D and taking the first d_1 rows of $(C_1D)(m)$, together with the first d_2 rows of $(C_2D)(m)$, ..., together with the first d_s rows of $(C_sD)(m)$ to construct a matrix. Since the choice of m and $d_1, \ldots, d_s$ was arbitrary, $C_1D, \ldots, C_sD$ generate a $(0, s)$-sequence in base q by Theorem 1. □

Remark 1. If in addition C_1 and C_1' are upper triangular matrices, then D is also upper triangular. Thus, for any $m > 0$ and point indices $0 \le i < q^m$, only the upper left $m \times m$-submatrix of D has an effect, compared to the complete m left columns if D was not upper triangular. Since $D(m)$ is regular, it is an automorphism of F_q^m and only reorders the indices. Therefore, the $(0, s)$-sequence in base q generated by $C_1, \ldots, C_s$ is identical to the one generated by $C_1D, \ldots, C_sD$, except for the numbering of the points $\mathbf{x}_0, \ldots, \mathbf{x}_{q^m-1}$ for all $m > 0$.

Modifying Sobol's $(0, 2)$-Sequence

In the remainder of this document, by $a \bmod m$ we mean the common residue. That is the nonnegative value $b < m$, such that $a \equiv b \pmod{m}$.

We now apply the theorem above to Sobol's $(0, 2)$-sequence. It is known that the infinite upper triangular matrices

$$C_1 := \left(\delta_{ij}\right)_{i,j=0}^{\infty}, \quad C_2 := \left(\binom{j}{i} \bmod 2\right)_{i,j=0}^{\infty}, \tag{5.14}$$

where δ_{ij} is the Kronecker delta, generate a $(0, 2)$-sequence in base $b = 2$ [Sob67]. It is identical to the Faure-sequence for $s = b = 2$ [Fau82]. The Larcher-Pillichshammer radical inverse in base $b = 2$ [LP01] is a $(0, 1)$-sequence given by the infinite generator matrix

$$C_1' := \left(\begin{cases} 1 & \text{if } i \leq j, \\ 0 & \text{otherwise} \end{cases}\right)_{i,j=0}^{\infty}. \tag{5.15}$$

Applying Theorem 2 with $D = C_1^{-1}C_1' = C_1'$ and

$$\begin{aligned} C_2' &:= C_2 D && (5.16)\\ &= \left(\binom{j}{i} \bmod 2\right)_{i,j=0}^{\infty} C_1' && (5.17)\\ &= \left(\sum_{k=0}^{j} \binom{k}{i} \bmod 2\right)_{i,j=0}^{\infty} && (5.18)\\ &= \left(\binom{j+1}{i+1} \bmod 2\right)_{i,j=0}^{\infty}, && (5.19) \end{aligned}$$

where the last equality follows from the Christmas Stocking Theorem [But02]:

$$\sum_{j=0}^{n} \binom{j}{k} = \binom{n+1}{k+1}. \tag{5.20}$$

By Theorem 2, the matrices C_1', C_2D generate a $(0, 2)$-sequence in base $b = 2$. As stated in [Nie92, Lemma 4.22, p. 62], adding the component $\frac{i}{2^m}$ for the i-th point, a $(0, m, 3)$-net can be constructed. The points generated this way are likely to perform better than the Sobol' $(0, m, 3)$-net when used for quasi-Monte Carlo integration as one of the projections is the Larcher-Pillichshammer $(0, m, 2)$-net with low star-discrepancy [LP01].

An area of further research is the difference of the smaller subnets that constitute larger subnets compared to the original sequence of Sobol'. By Remark 1 we know that each 2^m points of the modified sequence differ only in the numbering from the unmodified sequence. They therefore constitute the same $(0, m, 2)$-net. However, when using the subnets that constitute a larger net, the numbering might be important. [KK02b] used subnets of the Sobol' sequence for efficient trajectory splitting. The modified sequence might yield different results.

Remark 2. We could also have swapped the first two components of the Sobol' $(0, 2)$-sequence and consequently chosen $D = C_2^{-1}C_1'$. However, $C_2 = C_2^{-1}$, so $D = C_2C_1'$ and $C_2' = C_1D = D = C_2C_1'$, which is the exactly the same matrix as in the result above. $C_2 = C_2^{-1}$ because by Remark 1 the nets generated by C_1, C_2 must be identical to the nets generated by C_1C_2, C_2C_2, except of the numbering of the points. This implies $C_2C_2 = C_1$, so $C_2 = C_2^{-1}$.

Remark 3. We ran a combinatorial extensive search for upper triangular matrices that generate a $(0, 2)$-sequence together with the Larcher-Pillichshammer radical inverse generator matrix C_1'. The matrix size was limited in this search, but surprisingly we only found one single matching upper triangular matrix, namely the corresponding submatrix of C_2'. This indicates that all $(0, 2)$-sequences generated by upper triangular matrices produce the same $(0, m, 2)$-nets as the Sobol' $(0, 2)$-sequence.

Remark 4. Numerical experiments indicated that except for $m = 5$ and $m = 6$, the minimum toroidal distance achieved by $(0, m, 3)$-nets constructed using C_1', C_2' is by far superior to Sobol' $(0, m, 3)$-nets (see Figure 5.5).

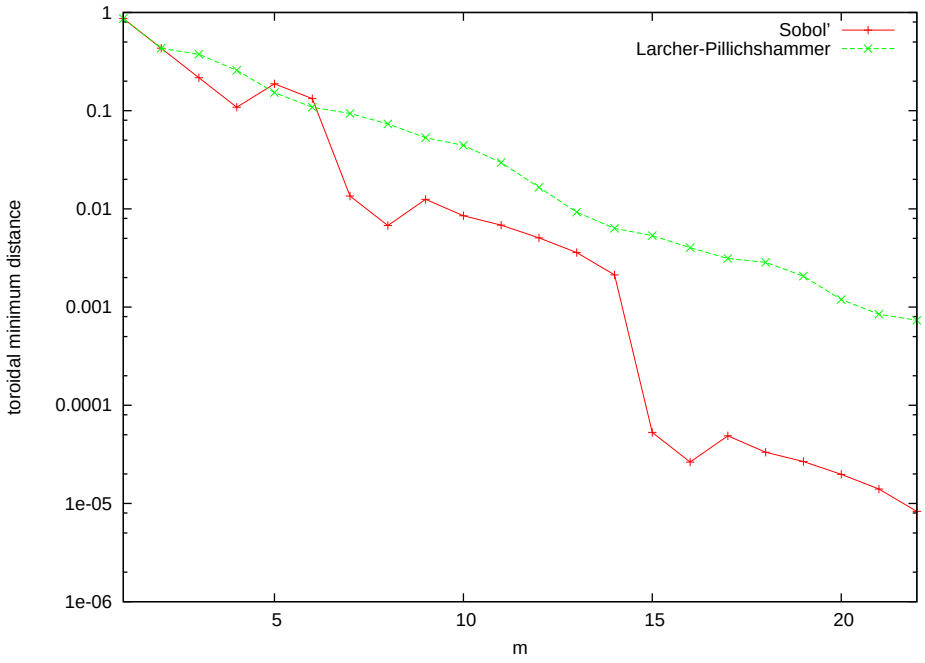


Figure 5.5: A plot of the toroidal distance in $[0,1)^3$ for both the Sobol' $(0, m, 3)$-net as well as the $(0, m, 3)$-net constructed using C_1', C_2', where one of the two-dimensional projections equals the Larcher-Pillichshammer point set. We would like to stress that the minimum distance is scaled logarithmically, so the absolute difference is very significant for increasing m. In contrast to the Sobol'-net, the minimum distance decreases smoothly for the new construction.

The minimum distance of these nets for $0 < m < 24$ has been computed using the following algorithm. Determining the minimum distance without a spatial structure is a typical $\mathcal{O}\left(n^2\right)$ problem for n points. Building spatial structures such as a grid [GHSK07] to determine the nearest neighbor is not an option for large m due to the excessive memory requirements.

However we can exploit the fact that the first coordinate of sample points in integer scale equals the sample index. Instead of enumerating all following indices for each point, we need to scan only those points within an index difference smaller than the currently determined minimum distance (possibly wrapping around the unit cube for the toroidal distance). In

other words, we only consider samples within a vertical stripe whose width equals the minimum distance. Sample points with other indices must be further away than the current minimum distance. [1]

This idea is implemented in the following parallelized C99 function with $\mathcal{O}(1)$ memory requirements. Special care is taken to avoid overflows due to the large numbers that occur when using accurate integer coordinates (where squared distances easily exceed 32 bits). The function pointer `f` points to a function that computes the y and z components of the $(0, m, 3)$-net.

```
inline unsigned long long computeMinDist(void (*f)(unsigned int,
unsigned int &, unsigned int &), const unsigned int m) {
  const int n = 1 << m;
  const unsigned int wrapMask = ~-n;
  const unsigned int half = n >> 1;

  // minimum distance is bounded by  3 * (n/2)^2
  unsigned long long minDist = 3 * ((1ULL << (m << 1)) >> 2);

#pragma omp parallel for schedule(dynamic)
  for (int i = 0; i < n; i++) {
    unsigned int iy, iz;
    f(i, iy, iz);
    iy >>= 32 - m;
    iz >>= 32 - m;

    for (unsigned int jj = i + 1; ; jj++) { // only scan as long as needed
      const unsigned int j = jj & wrapMask; // wrapped around

      unsigned int jy, jz;
      f(j, jy, jz);
      jy >>= 32 - m;
      jz >>= 32 - m;

      const unsigned int dx = jj - i;

      unsigned long long dist = (unsigned long long) dx * dx;
      if (dist >= minDist) // all remaining points cannot be nearer
        break;

      unsigned int dy = abs((int) (iy - jy));
      unsigned int dz = abs((int) (iz - jz));

      if (dy > half) // toroidal distance
        dy = n - dy;

      if (dz > half) // toroidal distance
```

[1] This could actually be further improved by only considering a disk around the current sample point instead of a stripe. The indices corresponding to the elementary intervals within the disk could be determined using the mappings from Subsection 5.2.4. However this would complicate the code quite a bit.

```
        dz = n - dz;

      dist += (unsigned long long) dy * dy + (unsigned long long) dz * dz;

      if (dist < minDist) // enter critical section only when needed
#pragma omp critical
        if (dist < minDist)
          minDist = dist;
    }
  }

  return minDist;
}
```

Remark 5. The following code in C99 returns the i-th point of the $(0, m, 3)$-net generated using C_1', C_2' for $m < 32$ in $\mathcal{O}(m)$. It relies on the fact that addition corresponds to XOR in $\mathbb{F}_2$.

```
void x_i(unsigned int i, const unsigned int m, float x[3]) {
  // first component: i / 2^m
  x[0] = (float) i / (1U << m);

  // remaining components by matrix multiplication
  unsigned int r1 = 0, r2 = 0;
  for (unsigned int v1 = 1U << 31, v2 = 3U << 30; i; i >>= 1) {
    if (i & 1) { // vector addition of matrix column by XOR
      r1 ^= v1;
      r2 ^= v2 << 1;
    }
    // update matrix columns
    v1 |= v1 >> 1;
    v2 ^= v2 >> 1;
  }

  // map to unit cube [0,1)^3
  x[1] = r1 * (1.f / (1ULL << 32));
  x[2] = r2 * (1.f / (1ULL << 32));
}
```

Modifying Faure's $(0, s)$-Sequences

Sometimes higher-dimensional $(0, s)$-sequences could be interesting in the domain of realistic rendering. For motion blur, three dimensions suffice: two for the sample position inside the pixel and one for the time. A three-dimensional sequence could be used for adaptive sample counts. After determining if the variance is high, more samples can be taken. The combination of the previous samples and the new samples would be well-stratified when using a $(0, 3)$-sequence. Depth of field simulation adds another two components and if area light sources are sampled for direct lighting, samples are 7-dimensional. It might be suffi-

cient to split up this 7-dimensional space into lower dimensional "slices", i.e. using separate low-dimensional samples that might not be well-stratified together, similar to [KK02a].

However, there exist a number of drawbacks when a base $b > 2$ is used.

- The generation of samples is much slower, since vector operations of binary bits cannot be used anymore. This means the matrix-vector multiplication of Equation 5.9 then takes $\mathcal{O}\left(m^3\right)$ instead of $\mathcal{O}\left(m^2\right)$ for the vectorized algorithm.
- Results cannot be accurately represented using binary floating point numbers anymore.
- To get from a $(0, m, s)$-net to a $(0, m+1, s)$-net, the sample count must be increased by factor $b > 2$. This stepping might be too coarse.

Nevertheless high-dimensional well-stratified sequences might be useful in some cases. The $(0, s)$-sequence by Faure [Fau82] is defined for any prime base $q \geq s$ and the generator matrices

$$C_i = \left(c_{jr}^{(i)}\right)_{j,r=0}^{\infty} \tag{5.21}$$

over F_q can be computed directly:

$$c_{jr}^{(i)} = \begin{cases} \binom{r}{j} a_i^{r-j} & \text{if } j \leq r, \\ 0 & \text{otherwise.} \end{cases} \tag{5.22}$$

where $a_1, \ldots, a_s$ denote distinct elements from F_q, e.g. $a_i = i - 1$.

Again we can change one of the generator matrices to the Larcher-Pillichshammer matrix to get superior $(0, m, s)$-nets. Interestingly, $C_i^{-1} = C_{s-i}$ when $a_i = i - 1$ is used, thus no matrix inversion algorithm is needed. To change C_i to the Larcher-Pillichshammer matrix L (see Equation 5.15), we simply multiply all matrices with $C_{s-i}L$.

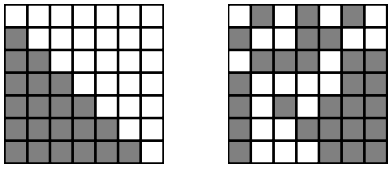
Limitations

Given generator matrices for a $(0, s)$-sequence, we are able to force one of the projections of the corresponding $(0, m, s+1)$-nets using Theorem 2. However, we are not able to modify two or more of the projections using this method. As soon as we multiply by another matrix in order to force a projection, all other matrices are changed.

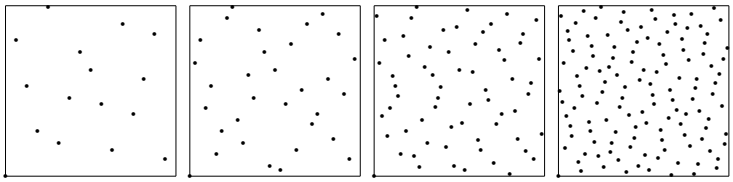
For example, it is not possible to force one component of Faure's $(0, 3)$-sequence in base 3 to be generated by the Larcher-Pillichshammer matrix, while at the same time another component is generated by the van der Corput identity matrix.

It is known that both the Larcher-Pillichshammer radical inverse and the van der Corput sequence have low discrepancy. Thus one could be tempted to try to find another $(0, s)$-sequence in base b where they can be used together. Unfortunately such a sequence cannot exist.

This follows from the more general version of Theorem 1, given in [Nie92, Thm. 4.36, p. 73]. For $m = 3$ we can combine the first row of the van der Corput matrix and the first two rows



(a) The boolean generator matrices (dark squares correspond to zeros).



(b) 16, 32, 64, and 128 points of the corresponding sequence.

Figure 5.6: 7×7 matrices that generate a $(0, 2)$-sequence with good two-dimensional minimum distance.

of the Larcher-Pillichshammer matrix:

$$C = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}. \tag{5.23}$$

By subtracting the first row from the second row of this matrix, it is obvious that C does not have full rank. Consequently, t must be larger than 0, independent on the base $b \geq 2$ and dimension $s \geq 2$ of the sequence. We would like to note that the Larcher-Pillichshammer points were originally only used for base 2, anyway. For base 2 there cannot exist $(0, s)$-sequences for $s \geq 3$ [Nie92, Cor. 4.24, p. 62].

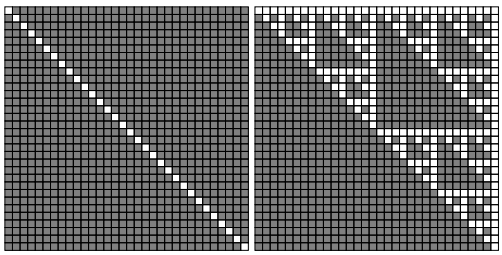
5.2.3 Other $(0, 2)$-Sequences

As described above, we found only a single matching *upper triangular* matrix that generates a $(0, 2)$-sequence together with the Larcher-Pillichshammer radical inverse. However there exist many other matrices of non-upper triangular shape that generate different $(0, 2)$-sequences (of finite precision) together with the Larcher-Pillichshammer generator matrix. Again we found these matrices using a backtracking computer search (however, this time without the restriction to upper triangular matrices).

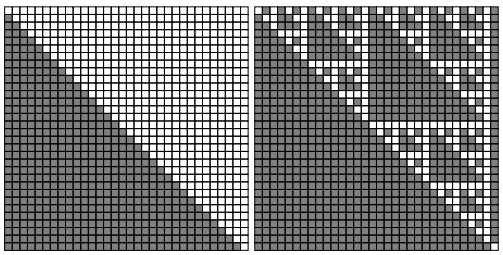
For most resulting matrices, a pattern of the matrix structure is not visible to the eye (e.g. Figure 5.6). Therefore a generalization to infinite precision is not possible from the finite-precision results of the computer search. Yet for some matrices such a pattern was visible, as depicted in Figure 5.7.

One can think of a number of optimization goals of the computer search, e.g.:

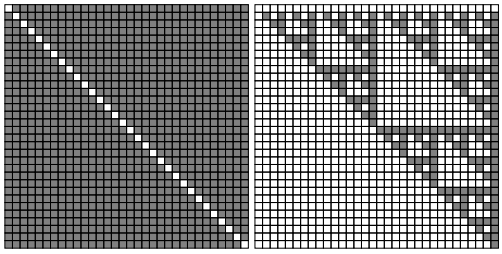
- Given a fixed component, find the best matrix within finite precision that generates a $(0, 2)$-sequence with maximized minimum distance. This minimum distance might



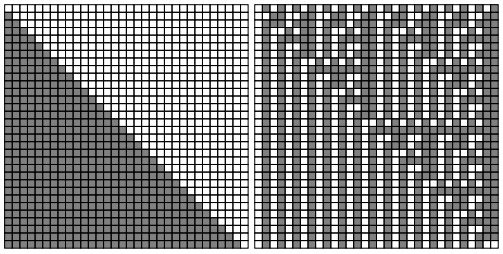
(a) Standard Sobol' $(0, 2)$-sequence.



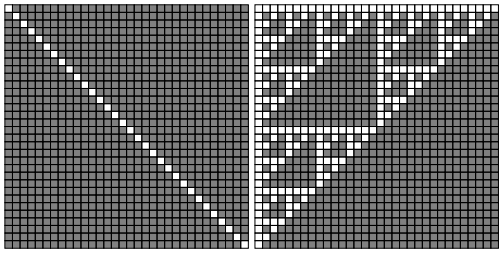
(b) Modified Sobol' $(0, 2)$-sequence to use the Larcher-Pillichshammer generator matrix.



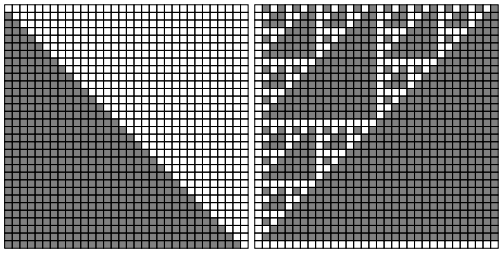
(c) A non-upper triangular matrix that generates a $(0, 2)$-sequence together with the van der Corput radical inverse.



(d) The matrix to the left modified to use the Larcher-Pillichshammer generator matrix.

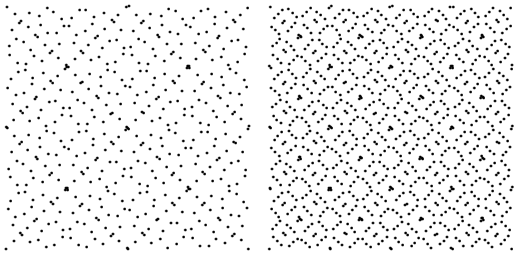


(e) Another non-upper triangular matrix that generates a $(0, 2)$-sequence together with the van der Corput radical inverse.

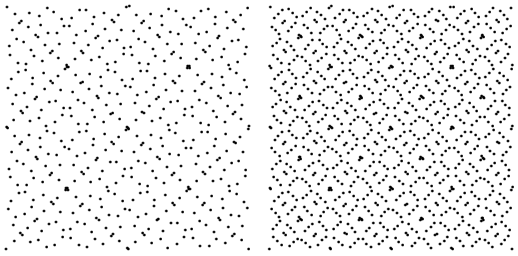


(f) The matrix to the left modified to use the Larcher-Pillichshammer generator matrix.

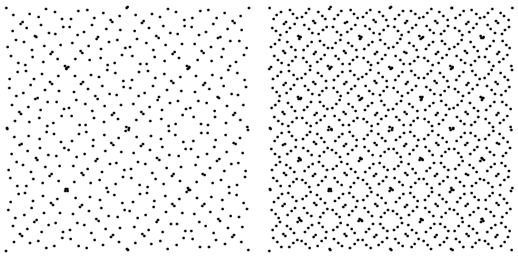
Figure 5.7: Different boolean $(0, 2)$-sequence matrices (dark squares correspond to zeros in the matrix). Corresponding points of the sequence are shown in Figure 5.8.



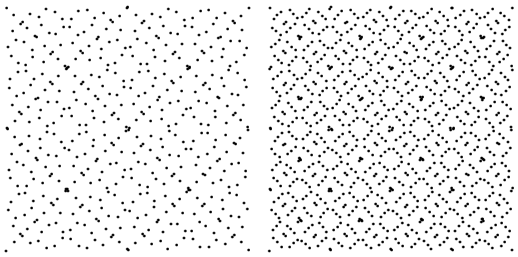
(a) Sobol' $(0, 2)$-sequence, 512 and 1024 points.



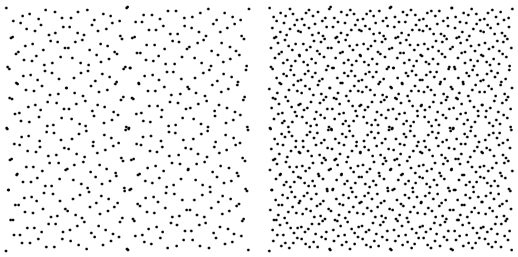
(b) The two-dimensional points do not change when the Larcher-Pillichshammer radical is used. However, the corresponding $(0, m, 3)$-net would be different.



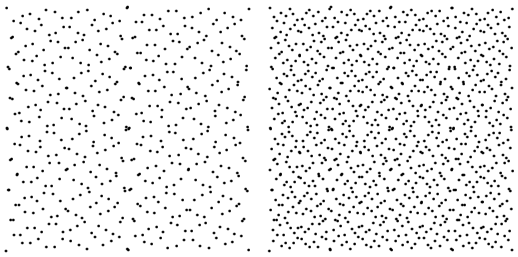
(c) 512 and 1024 points for the "inverse" Sobol' matrix.



(d) Like the left, but using the Larcher-Pillichshammer matrix. The points look slightly different since the second matrix was not upper-triangular.



(e) 512 and 1024 points for the "flipped" Sobol' matrix.



(f) Like the left, but using the Larcher-Pillichshammer matrix. The points look slightly different since the second matrix was not upper-triangular.

Figure 5.8: The corresponding points to the matrices shown in Figure 5.7.

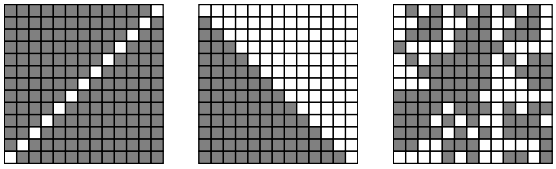


Figure 5.9: The matrices shown here generate a $(0, 13, 3)$-net in base 2 with a minimum distance of ≈ 0.01342385 (in comparison, the standard Sobol'-net achieves ≈ 0.00358812 and the modification with the Larcher-Pillichshammer generator matrix yields ≈ 0.00926449). Unfortunately for small values of m the other constructions are superior. The computer search quickly becomes intractable for large values of m.

be measured as a weighted sum of the distances for every 2^m points or as the distance for the maximal point set for the given precision.

- The same as above, but measuring distances of the corresponding $(0, m, 3)$-net by adding the $\frac{i}{n}$-component.

Sometimes such matrices can generate point sets with much larger minimum distance for certain numbers of points compared to the upper-triangular construction presented above. Figure 5.9 shows such a matrix together with the achieved minimum distance. However the computer search quickly becomes infeasible for larger values of m.

5.2.4 Mapping from Pixel to Sample Index

Instead of generating a well-stratified sampling pattern for every pixel, we can generate one large sampling pattern that is stratified over the whole screen [Kelo3]. Numerical experiments indicate that this approach often leads to faster convergence. One possible reason for this improved behavior might be the larger minimum distance that can be achieved when using patterns over the whole screen. Similar observations have been made for rank-1 lattices [Dam05]. Patterns over the whole screen are advantageous for another reason: Each pixel gets a different sampling pattern than its neighbours and therefore aliasing artifacts are reduced. The difference of having a sampling pattern per pixel vs. having a single sampling pattern over the whole screen can also be interpreted as the integration problem vs. the integro-approximation setting.

If we assume a framebuffer with a resolution of $2^m \times 2^m$ pixels, sampling patterns where the first two dimensions constitute a $(0, 2m, 2)$-net are ideal. These nets guarantee that exactly one sample falls into each pixel because one kind of elementary intervals is made of squares (see Figure 5.10). If more than one sample per pixel is desired, we simply map a higher resolution net to the pixel resolution.

Note that for high-definition image resolutions with a large number of samples per pixel, m may easily exceed 23. The resulting components of the points then cannot be accurately stored in 32-bit floating point numbers with a mantissa width of 23 bits (single-precision in the IEEE 754 Standard for Binary Floating-Point Arithmetic). Double-precision numbers should be accurate enough, though.

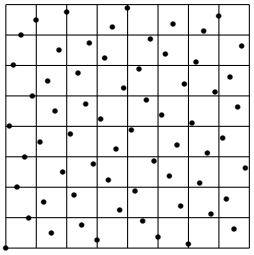


Figure 5.10: The use of a $(0, 2 \cdot 3, 2)$-net in base $b = 2$ guarantees that there is exactly one sample in each of the $2^3 \times 2^3$ pixels.

We could enumerate all samples and then determine the pixel index by multiplying the sample coordinates by 2^m and truncate to the nearest integer. However, the inverse mapping often is advantageous, i.e. to get the sample index for a given pixel index. This way, dividing the framebuffer into parts and assigning them to different processor cores is easy and makes parallelizing the rendering algorithm straightforward.

Additionally, enumerating samples near each other spatially increases the coherence of memory access when traversing the acceleration structure. This helps to reduce cache misses. Pixels (or tiles, consisting of equally sized rectangular areas of the image space) are often enumerated using a two-dimensional space-filling curve to improve spatial coherence. In practice often a Hilbert curve or the Z-order curve is used. The latter can be implemented very efficiently. Since the enumeration order of pixels is fixed by such an approach, we need to be able to retrieve the corresponding samples for each pixel.

Futhermore we generally do not have a framebuffer with $2^m \times 2^m$ pixels. For arbitrary resolutions, an appropriate $(0, 2m, 2)$-net can be used that is larger than the framebuffer. Instead of rejecting samples that do not fall into the framebuffer pixels, we can directly enumerate them using a mapping from pixel to sample index.

In [Kelo3], a method for generating such a mapping from pixel values to indices of a Hammersley $(0, 2m, 2)$-net is presented. It uses special properties of the van der Corput radical inverse sequence resulting in every line of the sampling pattern being a shifted copy of every other line as well as every column being a shifted copy of every other column. The shift is to be understood with a modulo wrap-around on the unit square, that is computing $\bmod 2^m$. While these properties cannot be assumed in general for matrix-generated $(0, 2m, 2)$-nets, the approach can be generalized.

The general construction of (t, m, s)-nets involves a matrix-vector multiplication. In the following, we assume that the first component of the samples is $\frac{i}{n}$, where i denotes the sample index and $n = 2^{2m}$. This corresponds to a generator matrix being the flipped unit matrix. We now concentrate on the generator matrix for the second coordinate.

The point index i is in the range $0, \ldots, 2^{2m} - 1$, thus it can be represented by using $2m$ bits. We split up this index into two parts, each consisting of m bits:

$$i = \sum_{j=0}^{2m-1} d_j(i) 2^j = \underbrace{\sum_{j=0}^{m-1} d_j(i) 2^j}_{m \text{ LSB}} + \underbrace{\sum_{j=m}^{2m-1} d_j(i) 2^j}_{m \text{ MSB}} \tag{5.24}$$

For base $b = 2$ and indices $k \cdot 2^m, \ldots, (k+1)2^m - 1$, only the m least significant bits (LSB) of the index change. These m bits correspond to the m leftmost columns of the generator matrix. Analogous, the m most significant bits (MSB) of each index correspond to the m rightmost matrix columns. That is why we precompute two tables, each of size 2^m. One table holds the results for the matrix-vector multiplications for all indices where the upper m bits are zero. The other table holds the results for indices where the lower m bits are zero. To find the result for an arbitrary index, we only have to split up the index into lower and upper m bits, look-up the results in the table and combine them with an exclusive-or operation.

Since the first coordinate of our samples is $\frac{i}{n}$, the requested pixel column directly determines the upper m bits. That is because each pixel has width 2^m in integer coordinates. We can therefore look up the appropriate table entry for the second coordinate given these upper m bits.

What remains is to find the lower m bits that let the sample fall into the pixel with the correct y-coordinate. We first use the m fixed upper index bits to look-up the corresponding table entry. The result will usually have m upper bits that differ from the ones we need to get the desired y-coordinate. The difference in these m upper bits of the result can be achieved by using the matching lower m index bits. The key of the algorithm is to precompute another table of size 2^m that holds an *inverse* mapping: given the upper m bits of the result we get the lower m index bits. Using this table we have all the $2m$ bits of the index and can easily compute the result.

The following code in C99 implements this idea and uses the Larcher-Pillichshammer radical inverse [KKo2b], but any other generator matrix in base $b = 2$ could be used. Note that the original procedure for the radical inverse does not need to be called anymore after the look-up tables have been computed. The parameters ex and ey enumerate the pixels, both run in the range $0, \ldots, 2^m - 1$. The resulting integer coordinates x and y need to be divided by 2^{2m} to yield values in $[0, 1)$.

```
// Larcher-Pillichshammer radical inverse in base 2 with 32 bits precision
inline unsigned int larcherPillichshammer(unsigned int i) {
  unsigned int r = 0;
  for (unsigned int v = 1U << 31; i; i >>= 1, v |= v >> 1)
    if (i & 1)
      r ^= v;
  return r;
}

void precomputeLookUpTables(const unsigned int m) {
  for (unsigned int i = 0; i < (1U << m); i++) {
    const unsigned int k = larcherPillichshammer(i) >> (32 - (m << 1));
    c1Lower[i] = k;
    c1LowerInverse[k >> m] = i; // inverse mapping
    c1Upper[i] = larcherPillichshammer(i << m) >> (32 - (m << 1));
  }
}

// looks up the sample for the square elementary interval (ex, ey).
// fills the integer coordinates (x, y) of the sample,
```

```
// where x is also the sample index.
inline void lookUp(const unsigned int m, const unsigned int ex,
const unsigned int ey, unsigned int &x, unsigned int &y) {
  const unsigned int delta = ey ^ (c1Upper[ex] >> m);
  const unsigned int lower = c1LowerInverse[delta];
  x = (ex << m) | lower;
  y = c1Upper[ex] ^ c1Lower[lower];
}
```

Without Look-Up Tables

Looking closely, the look-up tables are not necessary for upper triangular generator matrices, such as the Larcher-Pillichshammer matrix. Both the entries of `c1Lower` and `c1Upper` can be computed on-the-fly. The entries of `c1LowerInverse` seem more difficult, but by Theorem 1 the generator matrices are invertible. Denoting the Larcher-Pillichshammer matrix C_{LP}, this gives

$$C_{\mathrm{LP}} := \left(\begin{cases} 1 & \text{if } i \le j, \\ 0 & \text{otherwise} \end{cases} \right)_{i,j=0}^{\infty}, \text{with } C_{\mathrm{LP}}^{-1} := \left(\begin{cases} 1 & \text{if } i \le j \text{ and } j - i < 2, \\ 0 & \text{otherwise} \end{cases} \right)_{i,j=0}^{\infty}. \tag{5.25}$$

This leads to the following code that uses two function calls instead of the look-up table: one for the Larcher-Pillichshammer radical inverse and the other for the inverse mapping.

```
// inverse of Larcher-Pillichshammer radical inverse in base 2
// with 32 bits precision
inline unsigned int larcherPillichshammerInverse(unsigned int i) {
  unsigned int r = 0;
  for (unsigned int v = 3U << 30; i; i >>= 1, v >>= 1)
    if (i & 1)
      r ^= v;
  return r;
}

// computes the sample for the square elementary interval (ex, ey).
// fills the integer coordinates (x, y) of the sample,
// where x is also the sample index.
inline void compute(const unsigned int m, const unsigned int ex,
const unsigned int ey, unsigned int &x, unsigned int &y) {
  x = ex << m;
  y = ey << m;
  const unsigned int lpUpper = larcherPillichshammer(x) >> (32 - (m << 1));
  const unsigned int delta = ey ^ (lpUpper >> m);
  const unsigned int lower = larcherPillichshammerInverse(delta << (32 - m));
  x |= lower;
  y |= lpUpper & ~-(1 << m);
}
```

It probably depends on the architecture which approach is faster. The size of the precomputed tables is not very large, but nevertheless it might be advantageous to get by without additional memory. Furthermore, memory look-ups are quite expensive compared with arithmetic instructions on modern systems.

Scrambling

Incorporating scrambling [KKo2b] to achieve a digital shift [Lemo8, Sec. 5.2.2] is often helpful to generate (quasi-)random instances of the same net. Since addition in $\mathbb{F}_2$ corresponds to an exclusive-or operation, the implementation is very efficient. The following code takes two additional scrambling parameters as 32-bit integers. Since the sample coordinates now may have up to 32 bits precision (compared to $2m$ bits without scrambling), the resulting coordinates need to be divided by 2^{32} to scale them to $[0,1)^2$.

```
// computes the sample for the square elementary interval (ex, ey).
// fills the integer coordinates (x, y) of the scrambled sample
// and returns the sample index.
inline unsigned int compute(const unsigned int m,
const unsigned int ex, const unsigned int ey,
const unsigned int scramble_x, const unsigned int scramble_y,
unsigned int &x, unsigned int &y) {
  const unsigned int upper = (ex ^ (scramble_x >> (32 - m))) << m;
  const unsigned int lpUpper = larcherPillichshammer(upper) ^ scramble_y;
  const unsigned int mask = ~-(1 << m) << (32 - m); // m MSB
  const unsigned int delta = (ey << (32 - m)) ^ (lpUpper & mask);
  const unsigned int lower = larcherPillichshammerInverse(delta);
  const unsigned int index = upper | lower;
  x = (index << (32 - (m << 1))) ^ scramble_x;
  y = lpUpper ^ delta;
  return index;
}
```

More Components

This approach to map pixels to sample indices can also be used for the new $(0, m, 3)$-net that uses the Larcher-Pillichshammer radical inverse. Given the sample index, the third component can easily be computed using the modified Sobol' radical inverse. Of course more components of a (possibly scrambled) Sobol' sequence can be computed as well using the returned index. However the original Sobol' sequence generator matrices should be multiplied by D from the left, as described in Modifying Sobol's $(0, 2)$-Sequence. When $s > 3$ components are generated, the resulting (t, m, s)-net has a t-parameter larger than zero, though.

5.2.5 Progressive Integration by Padded Replications

Kollig and Keller developed so called "padded replications" of low-dimensional sample sets in the context of bidirectional path tracing [KKo2a]. The use of a shift on the unit torus generates a new set of samples while preserving the minimum distance and stratification

properties of the original set. This is not true for the elementary interval property. However, one can argue that shifting the sample set can be seen as shifting the integrand in the opposite direction while keeping the original samples.

Generating a padded replication of the $(0, 2m, 2)$-net that covers the whole screen can be useful for progressive rendering. Each new frame gets a new sample set which is equally well-stratified. After rendering the frame, the result is blended over the old frames to increase the total number of samples per pixel and thus the quality of anti-aliasing.

However, simply adding a random offset in $[0,1)^2$ and using a modulo wrap-around on the unit square is not sufficient. We could not guarantee anymore that exactly one sample falls into each pixel then. Because of that, we add a random shift that is a multiple of the extent of a single pixel. This way, there is exactly one sample per pixel, but the samples are still aligned on a $2^{2m} \times 2^{2m}$ grid. This can be ameliorated by adding a random jitter (j_x, j_y) with $j_x, j_y \in \left[0, 2^{-2m}\right)$ to the final floating point sample coordinate.

We would like to stress that these random offsets need to be computed only once per sample set. In the following code in C99, the random pixel shift is denoted by `psx` and `psy` both being integers in $[0, 2^m)$. The random floating point jitter offset is denoted by `jx`, `jy`, both in the range $[0,1)$.

```
for (unsigned int ey = 0; ey < (1U << m); ey++) {
  for (unsigned int ex = 0; ex < (1U << m); ex++) {
    // add pixel shift
    const unsigned int sx = (ex + psx) & ~-(1U << m); // mod 2^m
    const unsigned int sy = (ey + psy) & ~-(1U << m); // mod 2^m

    unsigned int i, j;
    lookUp(m, sx, sy, i, j);

    // subtract pixel shift to get back to pixel (ex, ey)
    i = (i - (psx << m)) & ~-(1U << (m << 1)); // mod 2^(2m)
    j = (j - (psy << m)) & ~-(1U << (m << 1)); // mod 2^(2m)

    // add jitter and map to [0,1)
    const double x = (double) (i + jx) / (1U << (m << 1));
    const double y = (double) (j + jy) / (1U << (m << 1));
  }
}
```

Figure 5.11 shows some instances using these random shifts and the resulting sample set when they are superimposed.

Instead of taking uniform random offsets we could also derandomize the algorithm by using deterministic low-discrepancy sequences like the Halton-sequence. This could increase convergence as the random offsets themselves would be well-stratified.

Mapping for the Sobol' $(0, 2)$-Sequence

While each sample set of the previous approach was well-stratified, the combination of the sample sets was not. That is because of the randomness of the chosen shifts. It therefore might

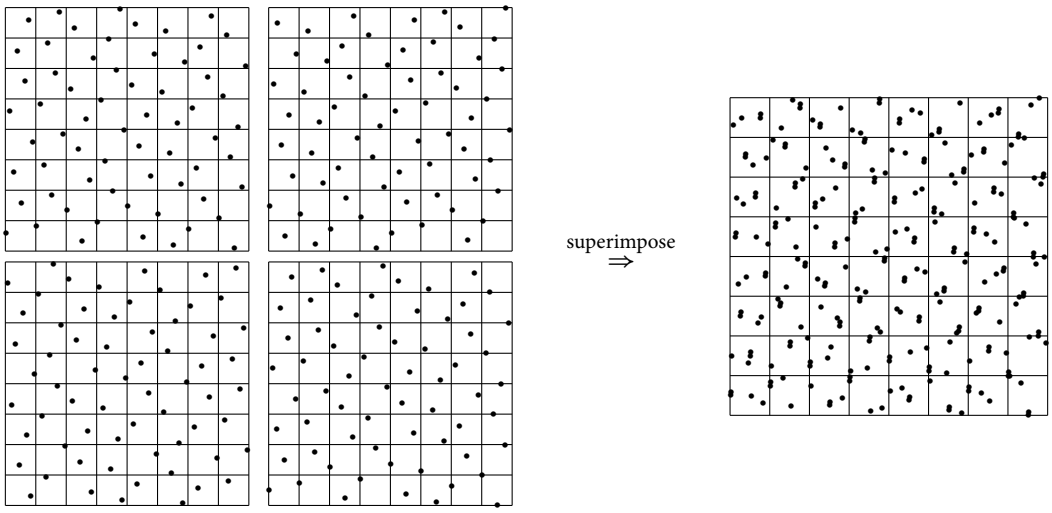


Figure 5.11: On the left, four different instances of the same sample set are depicted. For each instance, a different random shift was used. There always is exactly one sample in each pixel and each set is well-stratified over the whole screen. On the right, these four instances are superimposed and there are four samples per pixel. However, the samples can lie arbitrarily close together.

be beneficial to use the Sobol' $(0, 2)$-sequence. Its first component is given by the van der Corput radical inverse, its second by the Sobol' radical inverse [KK02b]. This complicates things somewhat as we cannot directly determine the m most significant bits of the index anymore. Instead, we need another table that stores the inverse for the van der Corput radical inverse.

```
// van der Corput radical inverse in base 2 with 32 bits precision
inline static unsigned int vanDerCorput(unsigned int bits) {
  bits = (bits << 16) | (bits >> 16);
  bits = ((bits & 0x00ff00ff) << 8) | ((bits & 0xff00ff00) >> 8);
  bits = ((bits & 0x0f0f0f0f) << 4) | ((bits & 0xf0f0f0f0) >> 4);
  bits = ((bits & 0x33333333) << 2) | ((bits & 0xcccccccc) >> 2);
  bits = ((bits & 0x55555555) << 1) | ((bits & 0xaaaaaaaa) >> 1);
  return bits;
}

// Sobol' radical inverse in base 2 with 32 bits precision
inline unsigned int sobol(unsigned int i) {
  unsigned int r = 0;
  for (unsigned int v = 1U << 31; i; i >>= 1, v ^= v >> 1)
    if (i & 1)
      r ^= v;
  return r;
}
```

```
void precomputeLookUpTables(const unsigned int m, const unsigned int frame) {
  const unsigned int shift = frame << (m << 1);

  for (unsigned int i = 0; i < (1U << m); i++) {
    const unsigned int j = vanDerCorput(i);
    c0Lower[i] = j;
    c0LowerInverse[j >> (32 - m)] = i; // inverse mapping
    c0Upper[i] = vanDerCorput((i << m) | shift);

    c1Lower[i] = sobol(i);
    const unsigned int k = sobol((i << m) | shift);
    c1Upper[i] = k;
    c1UpperInverse[k >> (32 - m)] = i; // inverse mapping
  }
}

// looks up the the sample for the square elementary interval (ex, ey).
// fills the 32-bit integer coordinates (x, y) of the sample,
// and returns the index of the sample.
inline unsigned int lookUp(const unsigned int m, const unsigned int frame,
const unsigned int ex, const unsigned int ey,
unsigned int &x, unsigned int &y) {
  const unsigned int lower = c0LowerInverse[ex];
  const unsigned int delta = ey ^ (c1Lower[lower] >> (32 - m));
  const unsigned int upper = c1UpperInverse[delta];
  const unsigned int index = ((upper << m) | lower) | (frame << (m << 1));
  x = c0Lower[lower] ^ c0Upper[upper];
  y = c1Lower[lower] ^ c1Upper[upper];
  return index;
}
```

The first two lines of the van der Corput function can be slightly optimized by using the `bswap` assembler instruction (e.g. by using `__bswap_32` defined in the `byteswap.h` header).

The look-up code above allows for specifying the current frame. This has the effect of adding an offset to all sample indices. The tables must be recomputed for each frame, but the resulting points will always constitute an $(0, 2m, 2)$-net. However, care must be taken not to discard bits since the integer coordinates of samples now can have more than $2m$ bits. Thus, the returned values of the look-up method must be divided by 2^{32} to map to $[0,1)^2$.

The returned index can be used to generate more components of the infinite dimensional Sobol' sequence. This is very useful for generating samples in the infinite dimensional path space for global illumination computations.

Figure 5.12 shows multiple frames with Sobol' nets and the resulting net when these smaller nets are superimposed. However, the distribution of a single set of Sobol' points is not as good as the Larcher-Pillichshammer points of the previous section.

We can get by with only one look-up table, similar to the approach at the end of Subsection 5.2.4. The van der Corput radical inverse generator matrix is the identity and therefore trivially its own inverse. Interestingly, the same is true for the Sobol' radical inverse generator matrix (see Remark 2 on p. 63). This means we can easily invert the Sobol' radical inverse, e.g.

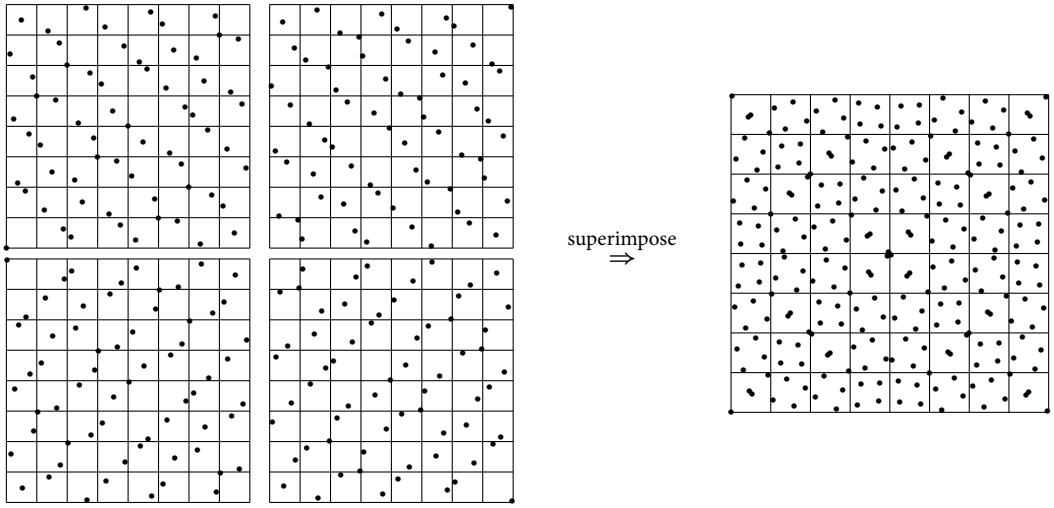


Figure 5.12: On the left, four different Sobol' $(0, 2m, 2)$-nets are depicted. There always is exactly one sample in each pixel and each set is well-stratified over the whole screen. On the right, these four instances are superimposed and there are four samples per pixel. The combination of points is better stratified than those of Figure 5.11.

using the following code:

```
// inverse of Sobol' radical inverse in base 2 with 32 bits precision
inline unsigned int sobolInverse(unsigned int i) {
  unsigned int r = 0;
  const unsigned int msb = 1U << 31;
  for (unsigned int v = 1; i; i <<= 1, v ^= v << 1)
    if (i & msb)
      r ^= v;
  return r;
}
```

However this does not help, since by using the van der Corput radical inverse for the first component, the lower m bits of the sample index are fixed. We then know how the upper m bits of the Sobol' component must look like to get to the correct y-coordinate. Yet we do not know the lower m bits of the Sobol' component and therefore cannot use the inversion above. That is why the following code to look-up the appropriate sample index still uses the Sobol' inversion table `c1UpperInverse`.

```
// returns the the sample index for the square elementary interval (ex, ey).
inline unsigned int lookUp(const unsigned int m, const unsigned int frame,
const unsigned int ex, const unsigned int ey) {
  // van der Corput is its own inverse
  const unsigned int lower = vanDerCorput(ex << (32 - m));
  const unsigned int delta = ey ^ (sobol(lower) >> (32 - m));
  const unsigned int upper = c1UpperInverse[delta];
  return ((upper << m) | lower) | (frame << (m << 1));
```

```
}
```

For $2m = 2^k$ we do not need any look-up tables. For any such m the Sobol' radical inverse generator matrix has the shape

$$\left(\begin{array}{c|c} A & A \\ \hline 0 & A \end{array}\right), \tag{5.26}$$

where A denotes an upper triangular matrix of size $m \times m$. In this case, we know how the lower m bits of the Sobol' component must look like. They are equal to the upper m bits! The following code implements this idea and uses four radical inverse function calls (apart from the shift that is constant per frame).

```
// fills the 32-bit integer coordinates (x, y) of the sample,
// and returns the index of the sample.
// important: this only works for m being a power of two!
inline unsigned int compute(const unsigned int m, const unsigned int frame,
const unsigned int ex, const unsigned int ey,
unsigned int &x, unsigned int &y) {
  // shift is constant per frame
  const unsigned int shift = frame << (m << 1);
  const unsigned int sobShift = sobol(shift);
  // van der Corput is its own inverse
  const unsigned int lower = vanDerCorput(ex << (32 - m));
  // need to compensate for ey difference and shift
  const unsigned int sobLower = sobol(lower);
  const unsigned int mask = ~-(1 << m) << (32 - m); // only m upper bits
  const unsigned int delta = ((ey << (32 - m)) ^ sobLower ^ sobShift) & mask;
  // only use m upper bits for the index (m is a power of two)
  const unsigned int sobResult = delta | (delta >> m);
  const unsigned int upper = sobolInverse(sobResult);
  const unsigned int index = shift | upper | lower;
  x = vanDerCorput(index);
  y = sobShift ^ sobResult ^ sobLower;
  return index;
}
```

The `frame` parameter could be used for adaptive sampling as well. When the number of samples per pixel is not known in advance, more samples can be generated by increasing `frame` for the same ex and ey values that correspond to the pixel. Of course the resulting set of pixel samples will not be as well stratified as using the same number of samples per pixel.

As an alternative to $(0, 2)$-sequences, the Halton sequence components $(\phi_2(i), \phi_3(i))$ could also be used. A direct mapping from pixels to the corresponding indices of the sequence is also possible, as described in [Kelo6].

5.3 Permutation-Generated (t, s)-Nets

Due to the Latin Hypercube stratification property of $(0, m, s)$-nets (the kind of elementary intervals at the very left and the very right in Figure 5.2 for the two-dimensional case), the

components of such a point set can be represented as permutations of each other. Interestingly, the construction of $(0,m,s)$-nets using generator matrices does not cover all such possible permutations that constitute a valid $(0,m,s)$-net. As we will see in this chapter, nets with increased minimum distance can be generated when all permutations are considered. These nets cannot be generated with the classic construction that uses generator matrices.

5.3.1 $(0,m,2)$-Nets in Base 2

All digital $(0,m,s)$-nets have a Latin Hypercube structure. This means the coordinates of the points are permutations of each other. To enumerate all possible $(0,m,2)$-nets we therefore consider all elements of the symmetric group S_n for $n = 2^m$. Such a permutation $\pi : \{0,\dots,n-1\} \to \{0,\dots,n-1\}$ then implies the point set $\{(i,\pi(i)) : i \in \{0,\dots,n-1\}\}$. Verifying the $t = 0$ property is possible using the code in given in [GHSK07, Section 2.2]. A mapping to the unit square $[0,1)^2$ can be achieved by dividing each component by n.

We have used Heap's efficient permutation generation algorithm [Sed77] to generate permutations and systematically search nets with maximized minimum distance. However, we did not create all permutations, but instead pruned the search tree in a backtracking style whenever the elementary-interval property of $(0,m,2)$-nets was violated or we had already found a net with larger minimum distance than the current one.

Despite this crucial optimization considering all possible permutations became intractable for $m > 6$ as $|S_n| = n! = (2^m)!$. However, we were able to find very regularly looking $(0,5,2)$-nets with a minimum distance of $\sqrt{32}/2^5 \approx 0.17677670$ [GHSK07], which is larger than the minimum distance of all $(0,5,2)$-nets generated in the classical way using generator matrices. This observation led to the following two constructions.

We would like to stress that in the components of the points are multiplied by 2^m, i.e. we will only consider the integer representation of points $(i,j) \in \{0,\dots,n-1\}^2$.

Construction for Odd m

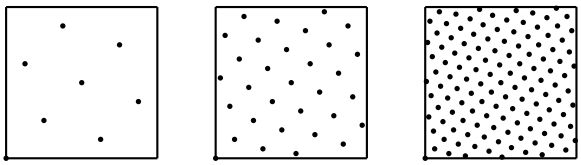


Figure 5.13: Examples of the permutation construction for $m = 3,5,7$ (from left to right).

Iterative Construction. There is a very simple iterative construction for this new kind of nets depicted in Figure 5.13. In this section we only conside odd values of m, generating a $(0,m,2)$-net with 2^m points. We start with the $(0,1,2)$-net given by the points $(0,0)$ and $(1,1)$. We then repeatedly quadruple the points until we have got 2^m points using the following rule. We assume we have a point set P with $|P| = 2^k$ points and want to yield 2^{k+2} points.

1. Lower left: $P_0 = \{(2i, 2j) : (i, j) \in P\}$.
2. Lower right: $P_1 = \left\{(2^{k+1} + 2i + 1, 2j + 1) : (i, j) \in P\right\}$.
3. Upper left: $P_2 = \left\{(2^{k+1} + 2i, 2^{k+1} + 2j) : (i, j) \in P\right\}$.
4. Upper right: $P_3 = \left\{(2^{k+2} + 2i + 1, 2^{k+1} + 2j + 1) : (i, j) \in P\right\}$.
5. The new point set then consists of $P' = P_0 \cup P_1 \cup P_2 \cup P_3$. Repeat this rule using P' as P until enough points have been created.

After enough iterations of this rule, we have a point set P with $|P| = 2^m$. We now need to "wrap" the resulting diagonals to yield points in $\{0, \dots, n-1\}^2$:

$$P' = \{(i \bmod 2^m, j) : (i, j) \in P\}. \tag{5.27}$$

The point set P' represents the final $(0, m, 2)$-net. This iterative construction is depicted in Figure 5.14.

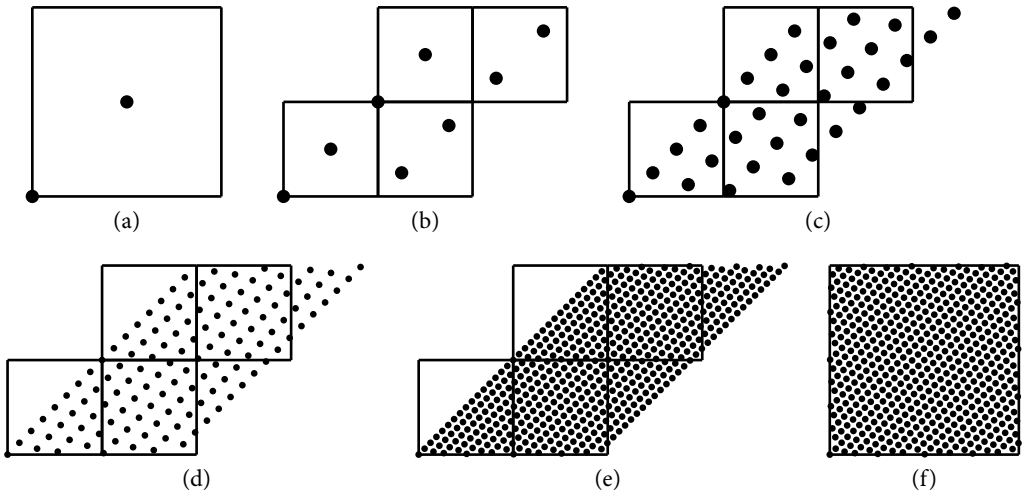


Figure 5.14: Iterative construction for odd m: The two initial points in (a) are quadrupled in each of the steps (b) – (e). Finally they are wrapped around to yield the $(0, 9, 2)$-net depicted in (f).

Direct Construction. As these nets consist of points lying on parallel diagonals similar to rank-1 lattices [DK07], a direct construction for a $(0, m, 2)$-net is possible by computing the "shifts" for each of the $2^{\lfloor \frac{m}{2} \rfloor}$ diagonals, each made up by $2^{\lceil \frac{m}{2} \rceil}$ points. If one point on each diagonal is known, all others result from it as they are evenly spaced along each diagonal by the offset vector $\left(2^{\lfloor \frac{m}{2} \rfloor}, 2^{\lfloor \frac{m}{2} \rfloor}\right)$. This diagonal is to be understood with a modulo n wrap-around for the first coordinate.

The position of the point with the smallest second coordinate on the k-th diagonal is given by $(2^m\phi_2(k) + k, k)$, where $0 \leq k < 2^{\lfloor \frac{m}{2} \rfloor}$ and $\phi_2(k) \in [0,1)$ denotes the van der Corput radical inverse in base 2.

Theorem 3. For odd m, the 2^m points $\left\{(x_{k,r}, y_{k,r}) : 0 \leq k < 2^{\lfloor \frac{m}{2} \rfloor}, 0 \leq r < 2^{\lceil \frac{m}{2} \rceil}\right\}$, where

$$x_{k,r} = (2^m\phi_2(k) + k + r \cdot 2^{\lfloor \frac{m}{2} \rfloor}) \bmod 2^m, \tag{5.28}$$

$$y_{k,r} = k + r \cdot 2^{\lfloor \frac{m}{2} \rfloor}, \tag{5.29}$$

constitute a $(0, m, 2)$-net in base 2 with a minimum distance of $\sqrt{2^m}$. This distance is measured using components multiplied by 2^m, i.e. on integer scale.

Proof. First, we will show that the elementary interval property holds. For each kind $0 \leq l \leq m$ of elementary intervals, there must be exactly one point in each elementary interval

$$\left[u \cdot 2^l, (u+1)2^l\right) \times \left[v \cdot 2^{m-l}, (v+1)2^{m-l}\right), \tag{5.30}$$

where $0 \leq u < 2^{m-l}$ and $0 \leq v < 2^l$. We consider the different kinds of elementary intervals separately. The different cases are depicted in Figure 5.15.

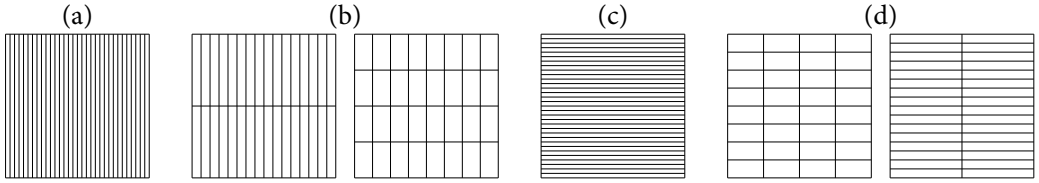


Figure 5.15: The different cases in the proof corresponding to certain kinds of elementary intervals, illustrated here for $m = 5$.

(a) $l = m$. There must be exactly one point in each horizontal stripe of width 2^m and height 1. Using Equation 5.29 and setting $v = y_{k,r}$, we can solve for k and r:

$$k = v \bmod 2^{\lfloor \frac{m}{2} \rfloor}, \tag{5.31}$$

$$r = \frac{v - k}{2^{\lfloor \frac{m}{2} \rfloor}}. \tag{5.32}$$

By the definition of the common residue it follows that $0 \leq k < 2^{\lfloor \frac{m}{2} \rfloor}$. Using Equation 5.31, the term $v - k$ is divisible by $2^{\lfloor \frac{m}{2} \rfloor}$. Since $0 \leq v < 2^l \leq 2^m$, we see that $0 \leq r < 2^{\lceil \frac{m}{2} \rceil}$. Consequently, there is a bijection from each of the 2^m horizontal stripes to the 2^m different parameter combinations (k, r).

(b) $\lceil \frac{m}{2} \rceil \leq l < m$. The width of these kinds of elementary intervals is greater than their height. We consider the y-coordinates $y_i := v \cdot 2^{m-l} + i$, where $0 \leq i < 2^{m-l} \leq 2^{m-\lceil \frac{m}{2} \rceil} =$

$2^{\lfloor \frac{m}{2} \rfloor}$. Again we can solve for the corresponding values for k and r:

$$k_i = y_i \bmod 2^{\lfloor \frac{m}{2} \rfloor} = (y_0 + i) \bmod 2^{\lfloor \frac{m}{2} \rfloor} = k_0 + i, \tag{5.33}$$

$$r_i = \frac{y_i - k_i}{2^{\lfloor \frac{m}{2} \rfloor}} = \frac{(y_0 + i) - (k_0 + i)}{2^{\lfloor \frac{m}{2} \rfloor}} = \frac{y_0 - k_0}{2^{\lfloor \frac{m}{2} \rfloor}}. \tag{5.34}$$

In other words, r_i is the same for all these points and the values k_i differ only in their least significant $m - l$ bits. The latter is true because the $m - l$ least significant bits of y_0 (and thus k_0) are zero and $i < 2^{m-l}$.

To prove that these points fall into different elementary intervals of width 2^l, we need to show that the $m - l$ most significant bits of their x-coordinates are different for each i. Considering Equation 5.28 and the reasoning above, we see that $r_i \cdot 2^{\lfloor \frac{m}{2} \rfloor}$ is the same for all i and the addition of k_i changes only the least $m - l \leq \lfloor \frac{m}{2} \rfloor$ bits. What remains is $2^m \phi_2(k_i) = 2^m \phi_2(k_0 + i)$. Since the $m - l$ least significant bits are different for each $0 \leq i < 2^{m-l}$ and ϕ_2 reverses the order of the bits, $2^m \phi_2(k_0 + i)$ differs in the $m - l$ most significant bits for each i.

(c) $l = 0$. There must be exactly one point in each vertical stripe of width 1 and height 2^m. The values $2^m \phi_2(k)$ are multiples of $2^{\lceil \frac{m}{2} \rceil}$ for each $0 \leq k < 2^{\lfloor \frac{m}{2} \rfloor}$. That is because the values of k differ in their $\lfloor \frac{m}{2} \rfloor$ least significant bits and all bit combinations are covered. Consequently, these values get mapped by the radical inverse ϕ_2 to multiples of $2^{-\lfloor \frac{m}{2} \rfloor}$. Thus we can solve for k and r by using Equation 5.28 and setting $u = x_{k,r}$:

$$k = u \bmod 2^{\lfloor \frac{m}{2} \rfloor}, \tag{5.35}$$

$$r = \frac{(u - 2^m \phi_2(k) - k) \bmod 2^m}{2^{\lfloor \frac{m}{2} \rfloor}}. \tag{5.36}$$

Analogous to the case $l = m$, by the definition of the common residue it follows that $0 \leq k < 2^{\lfloor \frac{m}{2} \rfloor}$. Using Equation 5.35, the term $u - k$ is divisible by $2^{\lfloor \frac{m}{2} \rfloor}$. With the reasoning above, $2^m \phi_2(k)$ is divisible by $2^{\lfloor \frac{m}{2} \rfloor}$ as well, resulting in $(u - 2^m \phi_2(k) - k) \bmod 2^m$ being divisible by $2^{\lfloor \frac{m}{2} \rfloor}$, and consequently $0 \leq r < 2^{\lceil \frac{m}{2} \rceil}$. Thus, there is a bijection from each of the 2^m vertical stripes to the 2^m different parameter combinations (k, r).

(d) $0 < l \leq \lfloor \frac{m}{2} \rfloor$. The height of these kinds of elementary intervals is greater than their width. We consider the x-coordinates $x_i := u \cdot 2^l + i$, where $0 \leq i < 2^l \leq 2^{\lfloor \frac{m}{2} \rfloor}$. Again we can solve for the corresponding values for k and r:

$$k_i = u_i \bmod 2^{\lfloor \frac{m}{2} \rfloor} = (u_0 + i) \bmod 2^{\lfloor \frac{m}{2} \rfloor} = k_0 + i, \tag{5.37}$$

$$r_i = \frac{(u_i - 2^m \phi_2(k_i) - k_i) \bmod 2^m}{2^{\lfloor \frac{m}{2} \rfloor}} \tag{5.38}$$

$$= \frac{((u_0 + i) - 2^m \phi_2(k_0 + i) - (k_0 + i)) \bmod 2^m}{2^{\lfloor \frac{m}{2} \rfloor}} \tag{5.39}$$

$$= \frac{(u_0 - 2^m \phi_2(k_0 + i) - k_0) \bmod 2^m}{2^{\lfloor \frac{m}{2} \rfloor}}. \tag{5.40}$$

Since the l least significant bits of x_0 (and thus k_0) are zero, only the l least significant bits differ of the values k_i.

To prove that these points fall into different elementary intervals of height 2^{m-l}, we need to show that the l most significant bits of their y-coordinates are different for each i. Considering Equation 5.29, the addition of k_i thus only changes the $l < \lfloor \frac{m}{2} \rfloor$ least significant bits, while the addition of $r_i \cdot 2^{\lfloor \frac{m}{2} \rfloor} = (u_0 - 2^m \phi_2(k_0 + i) - k_0) \bmod 2^m$ differs for each i only in the term $2^m \phi_2(k_0 + i)$. Since the l least significant bits are different for each $0 \leq i < 2^l$ and ϕ_2 reverses the order of the bits, $2^m \phi_2(k_0 + i)$ differs in the l most significant bits for each i.

We now consider the achieved minimum distance of the point set. Points on the diagonals are placed with a multiple of the offset $\left(2^{\lfloor \frac{m}{2} \rfloor}, 2^{\lfloor \frac{m}{2} \rfloor}\right)$, so their squared minimum distance to each other is

$$2 \cdot \left(2^{\lfloor \frac{m}{2} \rfloor}\right)^2 = 2^m. \tag{5.41}$$

The squared distance between the diagonals with slope 1 is

$$2 \cdot \left(\frac{2^{\lceil \frac{m}{2} \rceil}}{2}\right)^2 = 2^m. \tag{5.42}$$

In conclusion, the minimum distance is $\sqrt{2^m}$. As the diagonals can be tiled seamlessly, the result is identical for the toroidal distance measure. □

On the unit square $(0, 1]^2$, we need to divide the point coordinates by 2^m, so the minimum distance on the unit square is $\frac{\sqrt{2^m}}{2^m} = \sqrt{2^{-m}}$.

Inserting Equation 5.35 and Equation 5.36 into Equation 5.29 allows for another interpretation for the construction of these points. Given $u = x_{k,r}$,

$$y_{k,r} = k + \left(u - 2^m \phi_2\left(u \bmod 2^{\lfloor \frac{m}{2} \rfloor}\right) - k\right) \bmod 2^m \tag{5.43}$$

$$= \left(u - 2^m \phi_2\left(u \bmod 2^{\lfloor \frac{m}{2} \rfloor}\right)\right) \bmod 2^m. \tag{5.44}$$

This can be seen as replicating a $(0, \lfloor \frac{m}{2} \rfloor, 2)$ integer Hammersley-net $2^{\lceil \frac{m}{2} \rceil}$ times horizontally, scaling each one vertically by $-2^{\lceil \frac{m}{2} \rceil}$ and finally adding a linear component to the combination of nets. The resulting points are wrapped around the unit square.

Given the ability to compute the van der Corput radical inverse in base 2 [KKo2b], the computation of these permutation nets is possible in one line of code, following Equation 5.44:

```
// 32 bits van der Corput radical inverse in base 2
inline unsigned int phi2(unsigned int bits) {
  bits = (bits << 16) | (bits >> 16);
  bits = ((bits & 0x00ff00ff) << 8) | ((bits & 0xff00ff00) >> 8);
  bits = ((bits & 0x0f0f0f0f) << 4) | ((bits & 0xf0f0f0f0) >> 4);
  bits = ((bits & 0x33333333) << 2) | ((bits & 0xcccccccc) >> 2);
  bits = ((bits & 0x55555555) << 1) | ((bits & 0xaaaaaaaa) >> 1);
```

```
  return bits;
}

inline unsigned int y(const unsigned int u, const unsigned int m) {
  return (u - (phi2(u & ~-(1 << (m >> 1))) >> (32 - m))) & ~-(1U << m);
}
```

This can be made even faster for successive point requests by precomputing the bitmasks and the $2^{\lfloor \frac{m}{2} \rfloor}$ different radical inverse values.

Construction for Even m

For even $m \leq 6$ the permutation search did not reveal an improvement of the minimum distance in comparison to the full matrix search. We would like to note that the permutation search for $m = 6$ did not finish, though. However, when allowing a distance measure that accounts for a slightly irregular tiling, a very similar approach to the previous construction can be taken, yielding the nets shown in Figure 5.16.

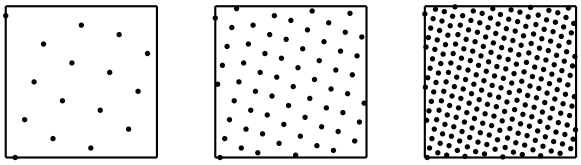


Figure 5.16: Examples of the permutation construction for $m = 4, 6, 8$ (from left to right).

Iterative Construction. This time, the iterative construction is a tad more involved. We start out with four diagonals, each consisting of four points:

$$\begin{aligned} P_0 &= \{(i+1, 4i) : \ i = 0, \ldots, 3\}, \\ P_1 &= \{(i+5, 4i+2) : \ i = 0, \ldots, 3\}, \\ P_2 &= \{(i+9, 4i+1) : \ i = 0, \ldots, 3\}, \\ P_3 &= \{(i+13, 4i+3) : \ i = 0, \ldots, 3\}. \end{aligned}$$

Then again we quadruple these points until we have reached the desired number of points, using the following rule. We assume we have a point set P with $|P| = 2^k$ points and want to yield 2^{k+2} points.

1. Multiply all points by two and add an offset for the points in P_0 and P_2:

$$P'_i = \{(2i + (i \bmod 2), 2j) : \ (i, j) \in P_i\}, \quad i = 0, \ldots, 3. \tag{5.45}$$

2. Extend the diagonals:

$$P''_i = P'_i \cup \left\{(2^{k-1} + i, 2^{k+1} + j) : \ (i, j) \in P'_i\right\}, \quad i = 0, \ldots, 3. \tag{5.46}$$

3. Combine diagonals:

$$P_0''' = P_0'' \cup P_1'', \tag{5.47}$$
$$P_1''' = P_2'' \cup P_3''. \tag{5.48}$$

4. Append shifted copies:

$$P_2''' = \left\{(2^{k+1} + i, j+1) : (i, j) \in P_0'''\right\}, \tag{5.49}$$
$$P_3''' = \left\{(2^{k+1} + i, j+1) : (i, j) \in P_1'''\right\}. \tag{5.50}$$

5. Repeat this rule using P_i''' as P_i until enough points have been created.

After enough iterations of this rule, we have a point set $P = P_0''' \cup P_1''' \cup P_2''' \cup P_3'''$ with $|P| = 2^m$. We now need to "wrap" the resulting diagonals to yield points in $\{0, \dots, n-1\}^2$:

$$P' = \{(i \bmod 2^m, j) : (i, j) \in P\}. \tag{5.51}$$

The point set P' represents the final $(0, m, 2)$-net. This iterative construction is depicted in Figure 5.17.

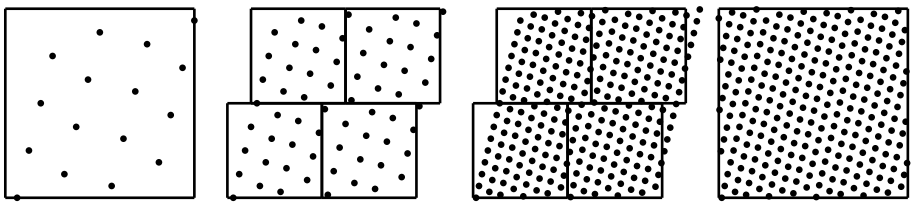


Figure 5.17: Iterative construction for even m: The 16 initial points in (a) are quadrupled in the steps (b) and (c). Finally they are wrapped around to yield the $(0, 8, 2)$-net depicted in (d).

Direct Construction. Using a tiling where each row of is shifted by $(2^{m-2}, 0)$ relative to its adjacent row below (see Figure 5.18), we can get a seamless tiling using $2^{\frac{m}{2}}$ diagonals, each with $2^{\frac{m}{2}}$ points. Again, the points are spaced evenly on each diagonal, this time by the offset vector $(2^{\frac{m}{2}-2}, 2^{\frac{m}{2}})$. The position of the point with the smallest second coordinate on the k-th diagonal, where $k \in \left\{0, \dots, 2^{\frac{m}{2}} - 1\right\}$, is given by $(2^m \phi_2(k) + \left\lfloor \frac{k}{4} \right\rfloor + 2^{\frac{m}{2}-2}, k)$. With the modified tiling described above, these diagonals are continued seamlessly.

These nets cannot be generated using the classical way of using generator matrices as the point $(0, 0)$ is not included.

We have verified the $t = 0$ property for all even $m \leq 22$. The minimum distance equals $\sqrt{241 \cdot 2^{m-8}}/2^m$ for all even m where $8 \leq m \leq 22$. See Table 5.1 for minimum distance values for $m < 8$. We would like to stress that the modified minimum distance measure that respects the shifts of the rows in the tiling has been used for these measurements.

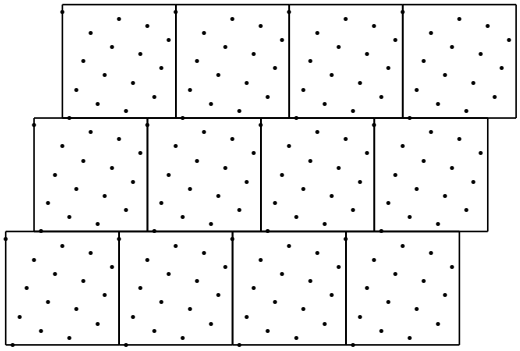


Figure 5.18: Modified tiling for the permutation construction where m is even and each row of the tiling is shifted by $(2^{m-2}, 0)$ relative to its adjacent row below.

On first sight it seems this pattern is only suited when we generate a sampling pattern for the whole screen since the special tiling does not match the arrangement of pixels on a common display. If we used the same pattern for each pixel, the minimum distance of the whole pattern would decrease. But unlike rank-1 lattices we do not need to compute a valid "shift" to guarantee that the same number of samples falls into each pixel for the pattern over the whole screen [Dam05]. Instead, this is guaranteed by the elementary intervals.

Looking at which samples fall into each pixel if we use the tiling of Figure 5.18, we arrive at Figure 5.19. Consequently, the sampling pattern for each pixel is actually the same, except for a modulo wrap-around in the direction of the x-axis that is dependent on the row. All pixels in a row share the same pattern.

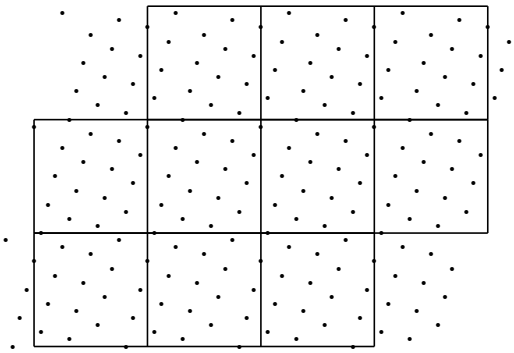


Figure 5.19: Shifting the tiling of Figure 5.18: All pixels in a row share the same sample pattern, while the pattern differs for pixels of neighboring rows by a modulo wrap-around in direction of the x-axis.

In other words, we can construct a larger pattern that consists of shifted copies of the original pattern, seen in Figure 5.20. There four rows and four columns of such modulo-wrapped patterns were combined to generate a larger pattern that matches the arrangement of the pixels on a screen. This pattern can be tiled regularly without a decrease in minimum distance.

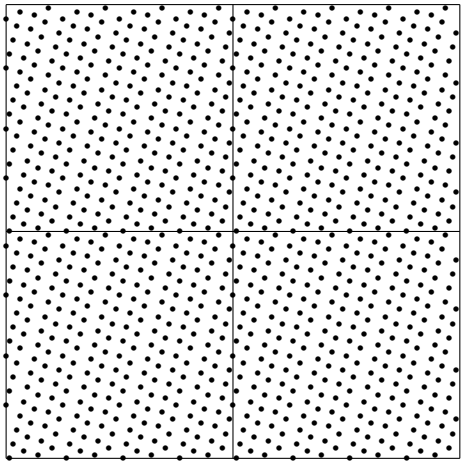


Figure 5.20: By taking 4×4 modulo-wrapped wrapped copies of the same pattern (see Figure 5.19), we get a larger pattern that is periodic and matches the pixel arrangement. It can be tiled regularly without a decrease in minimum distance.

Implementation for Arbitrary m

Exploiting the fact that we are generating nets in base 2, all operations needed to implement the permutation net constructions described above can be implemented very efficiently without any multiplications. The following C++ class generates the point sets directly for even and odd $m \geq 4$. It is comparable in terms of performance to generation of rank-1 lattice points.

```
typedef unsigned int uint;

class GenDiag0m2 {
  public:
    const uint m, n; // n = 2^m points are generated

    GenDiag0m2(const uint m)
    : m(m), n(1 << m), m2((m + 1) >> 1), mask(~-(1 << m2)) {
      const uint sqrtN = 1 << (m >> 1);
      d = new uint[sqrtN];

      if (m & 1) { // odd
        dx = dy = m >> 1;
        // precompute the diagonal shifts
        for (uint k = 0; k < sqrtN; k++)
          d[k] = (vdc(k) >> (32 - m)) + k;
      }
      else { // even
        dx = (m >> 1) - 2;
        dy = m >> 1;
        // precompute the diagonal shifts
        for (uint k = 0; k < sqrtN; k++)
          d[k] = (vdc(k) >> (32 - m)) + (k >> 2) + (1 << dx);
```

```
      }
    }

    ~GenDiagOm2() { delete [] d; }

    // returns the integer coordinates (x, y)
    // for the i-th point with i in {0, ..., n - 1}
    inline void operator()(const uint i, uint &x, uint &y) const {
      const uint k = i >> m2; // determine the diagonal
      const uint j = i & mask; // j-th point on the k-th diagonal

      // multiplication by shift, modulo by bitwise and
      x = (d[k] + (j << dx)) & (n - 1);
      y = k + (j << dy);
    }

  private:
    // 32 bits version of van der Corput radical inverse in base 2
    inline static uint vdc(uint bits) {
      bits = (bits << 16) | (bits >> 16);
      bits = ((bits & 0x00ff00ff) << 8) | ((bits & 0xff00ff00) >> 8);
      bits = ((bits & 0x0f0f0f0f) << 4) | ((bits & 0xf0f0f0f0) >> 4);
      bits = ((bits & 0x33333333) << 2) | ((bits & 0xcccccccc) >> 2);
      bits = ((bits & 0x55555555) << 1) | ((bits & 0xaaaaaaaa) >> 1);
      return bits;
    }

    const uint m2, mask;
    uint dx, dy, *d;
};
```

This class does not contain an implementation of the modified tiling for even m that was illustrated in Figure 5.19. The necessary modulo-wrap that depends on the pixel row is computed in the C++ code below. The resulting sample coordinates for the i-th sample of pixel (p_x, p_y) are integers x, y such that $p_x \cdot 2^m \leq x < (p_x+1)2^m$ and $(p_y \cdot 2^m \leq y < (p_y+1)2^m$. Note that this is only needed when m is even, since for odd m the regular tiling already matches the arrangement of pixels.

```
inline void pixelWrap(const GenDiagOm2 &gen, const uint p_x, const uint p_y,
                      const uint i, uint &x, uint &y) {
  gen(i, x, y);

  // modulo-wrap depending on row
  x += (p_y & 3) * (1 << (gen.m - 2));
  x &= ~-(1 << gen.m);

  // shift for pixel
  x += p_x << gen.m;
  y += p_y << gen.m;
}
```

Minimum Distance

A comparison of the toroidal minimum distance of different $(0, m, 2)$-net constructions can be found as a plot in Figure 5.21, with precise values given in Table 5.1. The permutation construction clearly features the largest minimum distance.

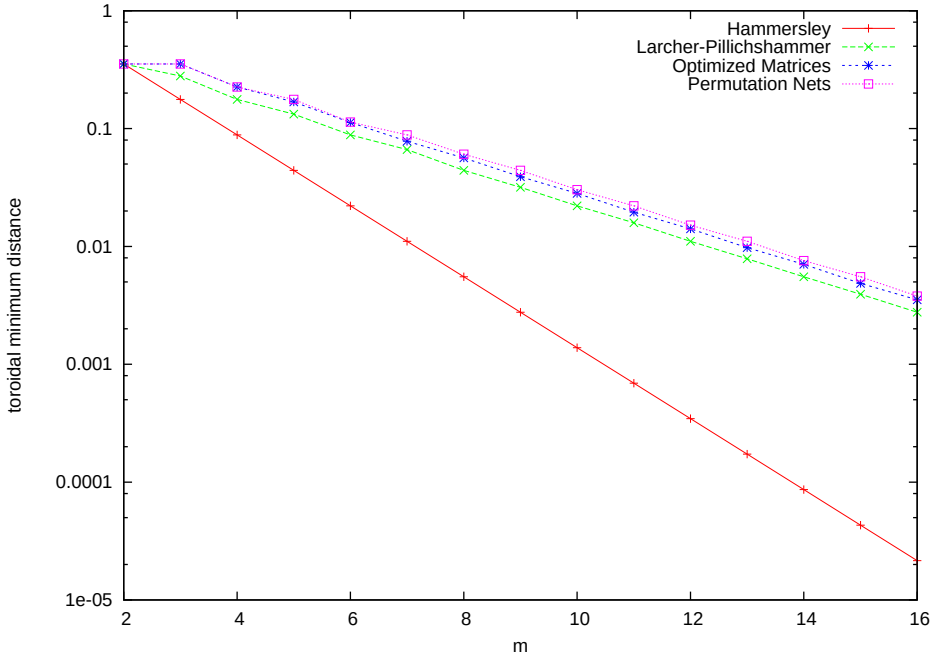
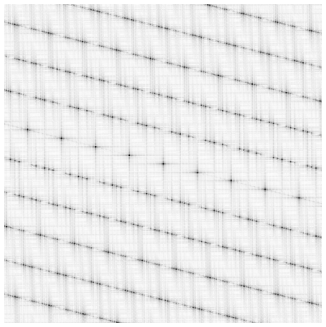


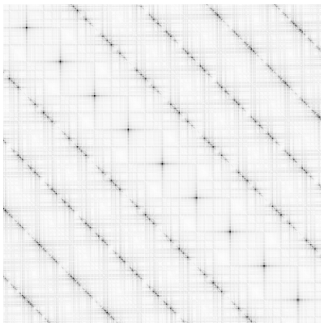
Figure 5.21: The minimum distances of different $(0, m, 2)$-nets. Please note that the y-axis is scaled logarithmically. Thus although the difference between the optimized matrices of [GHSK07] and the permutation nets does not seem that large in the plot, it is quite substantial.

Spectral Properties

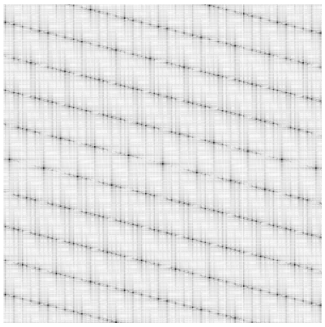
Unsurprisingly, the Fourier transformation for both even and odd m shows very pronounced diagonal structure (see Figure 5.22). This may lead to correlation artifacts in computer graphics applications, similar to rank-1 lattices, where the Fourier transformation is the dual lattice. While matrix-generated nets do not achieve the minimum distance of these permutation-based constructions, their spectrum looks more suitable for sampling in the domain of computer graphics.



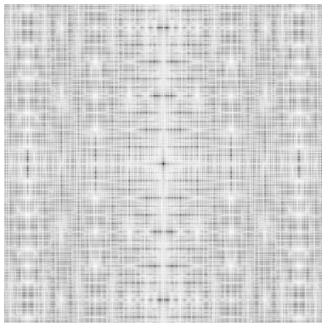
(a) $(0, 10, 2)$-permutation-net.



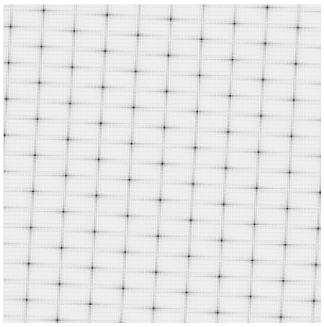
(b) $(0, 11, 2)$-permutation-net.



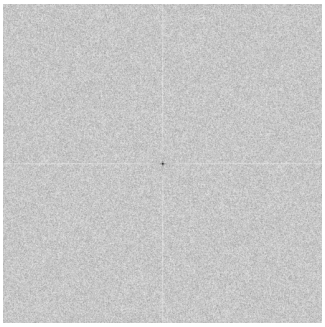
(c) 4×4 tiling of a modulo-wrapped $(0, 6, 2)$-net.



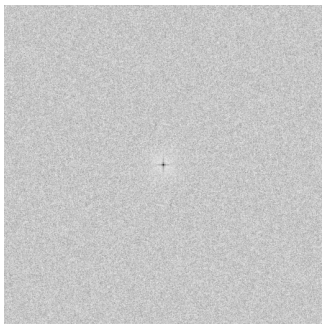
(d) Larcher-Pillichshammer-net [LP01].



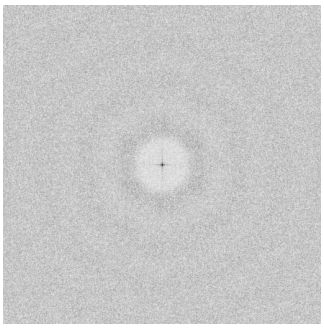
(e) Maximized minimum distance rank-1 lattice [DKD07].



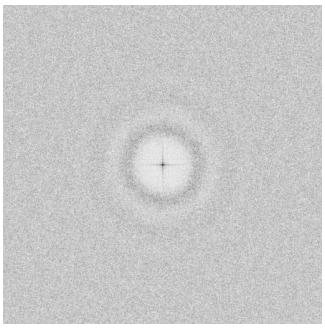
(f) n-rooks [PH04, Ch. 7].



(g) Jittered-grid [PH04, Ch. 7].



(h) Poisson-disk [HDK01].



(i) Lloyd relaxation [HDK01].

Figure 5.22: Fourier transformations of different two-dimensional sample patterns. Each pattern consisted of 1024 points, except for the $(0, 11, 2)$-permutation net with 2048 points.

m	Hammersley	Larcher-Pillichshammer	Opt. matrix [GHSK07]	Permutation
2	$\sqrt{2}/2^2 \approx 0.35355339$	$\sqrt{2}/2^2 \approx 0.35355339$	$\sqrt{2}/2^2 \approx 0.35355339$	$\sqrt{2}/2^2 \approx 0.35355339$
3	$\sqrt{2}/2^3 \approx 0.17677670$	$\sqrt{5}/2^3 \approx 0.27950850$	$\sqrt{8}/2^3 \approx 0.35355339$	$\sqrt{8}/2^3 \approx 0.35355339$
4	$\sqrt{2}/2^4 \approx 0.08838835$	$\sqrt{8}/2^4 \approx 0.17677670$	$\sqrt{13}/2^4 \approx 0.22534695$	$\sqrt{13}/2^4 \approx 0.22534695$
5	$\sqrt{2}/2^5 \approx 0.04419417$	$\sqrt{18}/2^5 \approx 0.13258252$	$\sqrt{29}/2^5 \approx 0.16828640$	$\sqrt{32}/2^5 \approx 0.17677670$
6	$\sqrt{2}/2^6 \approx 0.02209709$	$\sqrt{32}/2^6 \approx 0.08838835$	$\sqrt{52}/2^6 \approx 0.11267348$	$\sqrt{53}/2^6 \approx 0.11375172$
7	$\sqrt{2}/2^7 \approx 0.01104854$	$\sqrt{72}/2^7 \approx 0.06629126$	$\sqrt{100}/2^7 \approx 0.07812500$	$\sqrt{128}/2^7 \approx 0.08838835$
8	$\sqrt{2}/2^8 \approx 0.00552427$	$\sqrt{128}/2^8 \approx 0.04419417$	$\sqrt{208}/2^8 \approx 0.05633674$	$\sqrt{241}/2^8 \approx 0.06064131$
9	$\sqrt{2}/2^9 \approx 0.00276214$	$\sqrt{265}/2^9 \approx 0.03179457$	$\sqrt{400}/2^9 \approx 0.03906250$	$\sqrt{512}/2^9 \approx 0.04419417$
10	$\sqrt{2}/2^{10} \approx 0.00138107$	$\sqrt{512}/2^{10} \approx 0.02209709$	$\sqrt{832}/2^{10} \approx 0.02816837$	$\sqrt{964}/2^{10} \approx 0.03032065$
11	$\sqrt{2}/2^{11} \approx 0.00069053$	$\sqrt{1060}/2^{11} \approx 0.01589729$	$\sqrt{1600}/2^{11} \approx 0.01953125$	$\sqrt{2048}/2^{11} \approx 0.02209709$
12	$\sqrt{2}/2^{12} \approx 0.00034527$	$\sqrt{2048}/2^{12} \approx 0.01104854$	$\sqrt{3328}/2^{12} \approx 0.01408418$	$\sqrt{3856}/2^{12} \approx 0.01516033$
13	$\sqrt{2}/2^{13} \approx 0.00017263$	$\sqrt{4153}/2^{13} \approx 0.00786667$	$\sqrt{6385}/2^{13} \approx 0.00975417$	$\sqrt{8192}/2^{13} \approx 0.01104854$
14	$\sqrt{2}/2^{14} \approx 0.00008632$	$\sqrt{8192}/2^{14} \approx 0.00552427$	$\sqrt{13312}/2^{14} \approx 0.00704209$	$\sqrt{15424}/2^{14} \approx 0.00758016$
15	$\sqrt{2}/2^{15} \approx 0.00004316$	$\sqrt{16612}/2^{15} \approx 0.00393334$	$\sqrt{25313}/2^{15} \approx 0.00485536$	$\sqrt{32768}/2^{15} \approx 0.00552427$
16	$\sqrt{2}/2^{16} \approx 0.00002158$	$\sqrt{32768}/2^{16} \approx 0.00276214$	$\sqrt{53248}/2^{16} \approx 0.00352105$	$\sqrt{61696}/2^{16} \approx 0.00379008$

Table 5.1: Minimum distance of $(0, m, 2)$-nets in base 2 for Hammersley, Larcher-Pillichshammer, the construction for optimized matrices given in [GHSK07] and the permutation construction. These values are plotted in Figure 5.21.

Full Owen-Scrambling

In order to allow for the simple error estimate of Monte Carlo methods when using quasi-Monte Carlo integration, the point sets can be randomized. One particular powerful randomization method known as Owen-scrambling was presented in [Owe95] and analyzed in [Owe97] that allows for a variance estimation. The resulting points are uniformly distributed and therefore allow for an unbiased estimate. On the other hand they still form a (t, m, s)-net (or a (t, s)-sequence) with with probability 1.

The basic idea of this randomization is to use permutations to scramble the digits of the points in base b. More precisely, let $a_{ijk} \in \{0, \dots, b-1\}$ denote the k-th digit of the j-th component for the i-th point, i.e. the point $\mathbf{A}_{ij} \in [0,1)^s$ can be written as

$$\mathbf{A}_{ij} = \sum_{k=1}^{\infty} b^{-k} a_{ijk}. \tag{5.52}$$

Then the corresponding scrambled point $\mathbf{X}_{ij}$ is defined as

$$\mathbf{X}_{ij} = \sum_{k=1}^{\infty} b^{-k} x_{ijk}, \tag{5.53}$$

where x_{ijk} is a permutation of the digit a_{ijk}:

$$x_{ijk} = \pi_{j \cdot a_{ij1} \dots a_{ij(k-1)}}(a_{ijk}). \tag{5.54}$$

Here $\pi_{j \cdot a_{ij1} \dots a_{ij(k-1)}}$ denotes a random uniform permutations of $0, \dots, b-1$, with all permutations being indepent. The permutation applied to the k-th digits of $\mathbf{A}_{ij}$ therefore depends on both the component j and the preceding $k-1$ digits.

Since $\mathcal{O}(sb^m)$ permutations are needed for this scrambling, the permutations generally cannot be computed beforehand. [FK02] describes a method that allows for an efficient traversal of the permutation tree with low memory requirements. The following code is based on this approach and implements Owen-scrambling optimized for base $b = 2$ for the $(0, m, 2)$-permutation-net construction presented above. Some different random instances are depicted inFigure 5.23.

```
const unsigned int n = 1U << m;

unsigned int *points0 = new unsigned int[n]; // first component
unsigned int *points1 = new unsigned int[n]; // second component

GenDiag0m2 gen(m);

// generate inverse permutation
unsigned int *invPhi = new unsigned int[n];
for (unsigned int k = 0; k < n; k++) {
  unsigned int i, j;
  gen(k, i, j);
  invPhi[i] = j;
}
```

```
// generate fully Owen-scrambled point set
unsigned int component0 = get_rand_uint32();
points0[0] = component0;

unsigned int component1 = get_rand_uint32();
points1[invPhi[0]] = component1;

for (unsigned int i = 1; i < n; i++) {
  // detect where permutation tree ramifies:
  // find the bit where (i - 1) and i differ.
  unsigned int diff = (i - 1) ^ i, nEqualBits = 0;
  for (; diff; nEqualBits++, diff >>= 1);
  const unsigned int shift = m - nEqualBits;

  // change the branch and choose new random permutations
  component0 ^= ((1ULL << 31) | get_rand_uint32()) >> shift;
  points0[i] = component0;

  component1 ^= ((1ULL << 31) | get_rand_uint32()) >> shift;
  points1[invPhi[i]] = component1;
}

for (unsigned int i = 0; i < n; i++) {
  const double x = (double) points0[i] / (1ULL << 32);
  const double y = (double) points1[i] / (1ULL << 32);
  cout << x << " " << y << endl;
}
```

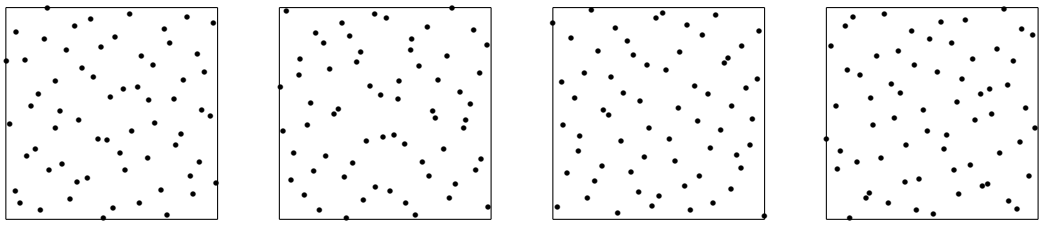


Figure 5.23: Four different instances of the same $(0, 6, 2)$-permutation-net created using different seeds for the random number generator to implement Owen-scrambling. The resulting points are uniformly distributed but are still stratified as a $(0, 6, 2)$-net. However the previous minimum distance property does not hold anymore. In contrast a Cranley-Patterson rotation generally yields point sets with the same toroidal minimum distance, but the (t, m, s)-net property gets violated.

5.3.2 $(0, m, 3)$-Nets in Base 2

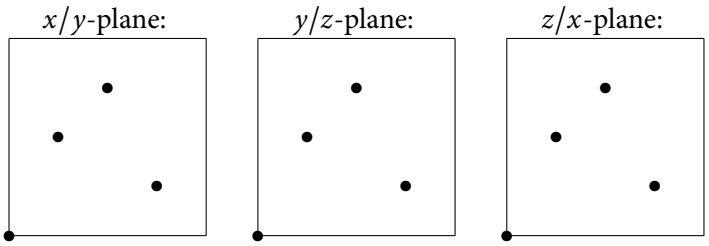
The permutation construction can easily be generalized to higher dimensions. Unfortunately for 2^m points in s dimensions the number of permutations is $((2^m)!)^{s-1}$ and therefore be-

comes intractable in the three-dimensional case for $m \geq 5$.

Nevertheless, the maximized minimum distance search for $s = 3$ and $m < 5$ yielded some interesting results: Analogous to the two-dimensional case, for some values of m, the best nets cannot be represented using the classical generator matrix construction. This is obvious since these optimal point sets do not include the origin $(0, 0, 0)$. Consequently, the maximum possible minimum distance for digital nets cannot be achieved using generator matrices. In the following we present some instances of optimal three-dimensional permutation nets, with permutations P_m and two-dimensional projections to the cardinal planes.

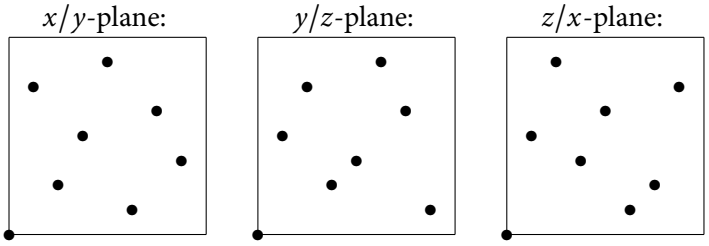
- $m = 2$. Minimum distance: $\sqrt{6}/2^2 \approx 0.6123724358$.

$$P_2 = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 2 & 3 & 1 \\ 0 & 3 & 1 & 2 \end{pmatrix}$$



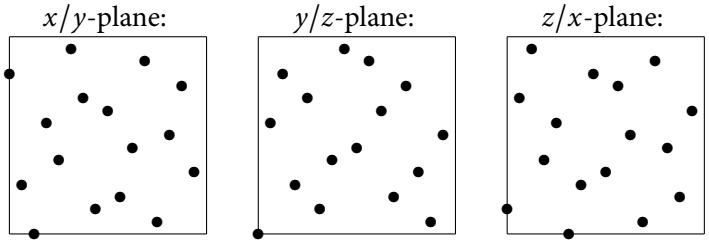
- $m = 3$. Minimum distance: $\sqrt{12}/2^3 \approx 0.4330127019$.

$$P_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 6 & 2 & 4 & 7 & 1 & 5 & 3 \\ 0 & 5 & 6 & 3 & 1 & 4 & 7 & 2 \end{pmatrix}$$



- $m = 4$. Minimum distance: $\sqrt{38}/2^4 \approx 0.3852758752$.

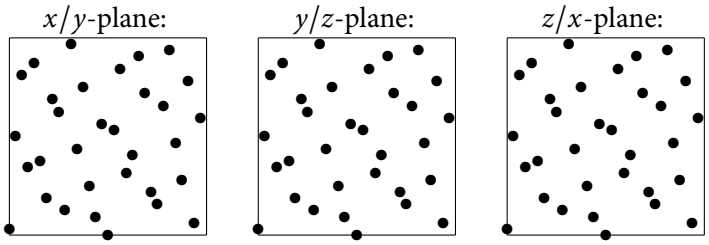
$$P_4 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 13 & 4 & 0 & 9 & 6 & 15 & 11 & 2 & 10 & 3 & 7 & 14 & 1 & 8 & 12 & 5 \\ 5 & 11 & 0 & 14 & 6 & 8 & 3 & 13 & 10 & 4 & 15 & 1 & 9 & 7 & 12 & 2 \end{pmatrix}$$



- $m = 5$. Minimum distance: $\sqrt{90}/2^5 \approx 0.2964635307$.
  Here we restricted the search to yield identical two-dimensional projections. This result thus is probably *not optimal*.

$$P_5 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 1 & 16 & 26 & 11 & 28 & 12 & 6 & 22 & 20 & 4 & 31 & 14 & 24 & 8 & 3 & 18 \\ 16 & 0 & 30 & 14 & 9 & 24 & 6 & 23 & 13 & 28 & 19 & 3 & 5 & 20 & 11 & 27 \end{pmatrix}$$

$$\begin{pmatrix} 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 0 & 17 & 27 & 10 & 13 & 29 & 23 & 7 & 5 & 21 & 30 & 15 & 9 & 25 & 2 & 19 \\ 1 & 17 & 15 & 31 & 8 & 25 & 7 & 22 & 12 & 29 & 2 & 18 & 4 & 21 & 26 & 10 \end{pmatrix}$$



Since 2^5 points are not numerous enough for most practical applications, we propose the following three-dimensional tiling that preserves the Latin Hypercube structure. We assume that 2^m three-dimensional points are given in integer coordinates. This tiling then yields 2^{m+3} points and can be repeated as often as desired.

1. Multiply all coordinates by four.
2. Place a copy of the point set into each octant, using the following shifts:

Octant	Δx	Δy	Δz
000	0	0	0
001	2	2	2^{m+2}
010	1	2^{m+2}	2
011	3	$2 + 2^{m+2}$	$2 + 2^{m+2}$
100	2^{m+2}	1	3
101	$2 + 2^{m+2}$	3	$3 + 2^{m+2}$
110	$1 + 2^{m+2}$	$1 + 2^{m+2}$	1
111	$3 + 2^{m+2}$	$3 + 2^{m+2}$	$1 + 2^{m+2}$

6 Numerical Results

We used different point sets, including the new constructions, to render test scenes using a varying number of samples. Given a reference image I_{ref}, the error of another image I was measured as $\|I - I_{\text{ref}}\|_2$, where an image I is interpreted as a vector consisting of the color values of the pixels.

6.1 Sampling in Dimension $s = 2$

We have used two different classic test scenes to measure the convergence using different two-dimensional sample points: A perspective checkerboard view and concentric sinus-modulated circles. The following samplers have been used to generate points in $[0,1)^2$:

- Jittered grid.
- Maximized minimum distance rank-1 lattice [DKD07], using a Cranley-Patterson rotation to avoid correlation with lines present in the image.
- Hammersley $(0, m, 2)$-net, stratified over the whole screen.
- Larcher-Pillichshammer $(0, m, 2)$-net, stratified over the whole screen.
- Optimized matrix-generated $(0, m, 2)$-nets (in the sense of maximizing minimum distance) [GHSK07], stratified over the whole screen.
- Permutation-generated nets, stratified over the whole screen.

The following figures show images that have been computed using 64 samples per pixel (except for the reference images with 125 000 samples per pixel, computed using jittered grids of size 500^2). The image on the right of each figure depicts a magnified area of the image. A convergence plot for different sample counts can be found at the end of each section for both test scenes.

6.1.1 Checkerboard

This scenes features a classic checkerboard view. The results of the different sampling strategies are shown in Figure 6.1, with the corresponding convergence graphs in Figure 6.2.

The quasi-Monte Carlo samplers perform similarly well for this scene, with the Larcher-Pillichshammer points yielding the best results for most sample counts. The rank-1 lattice performance depends very much on the sample count, probably because of correlations between the generator vector and the checkerboard pattern.

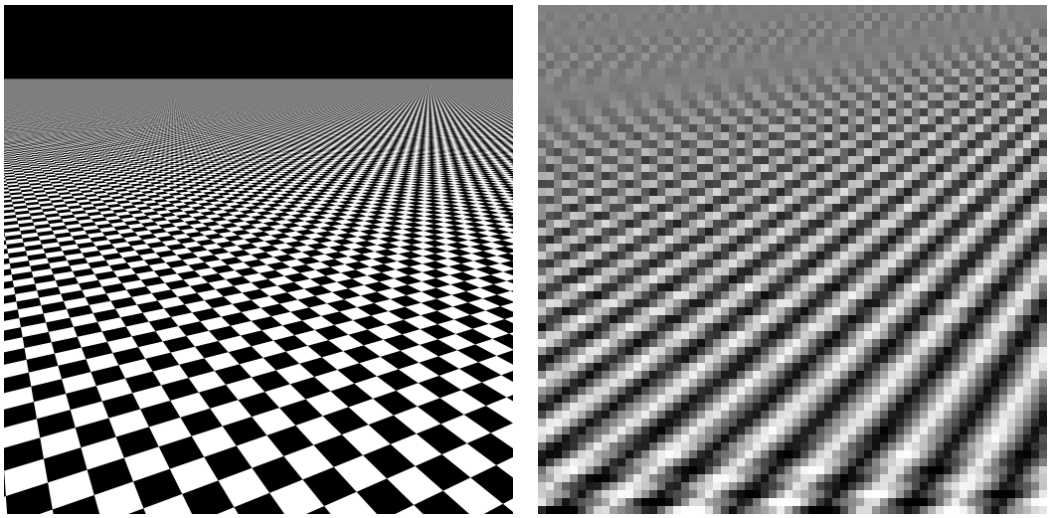
6.1.2 Zone Plate Test Pattern

This scene tests anti-aliasing using the zone plate test pattern

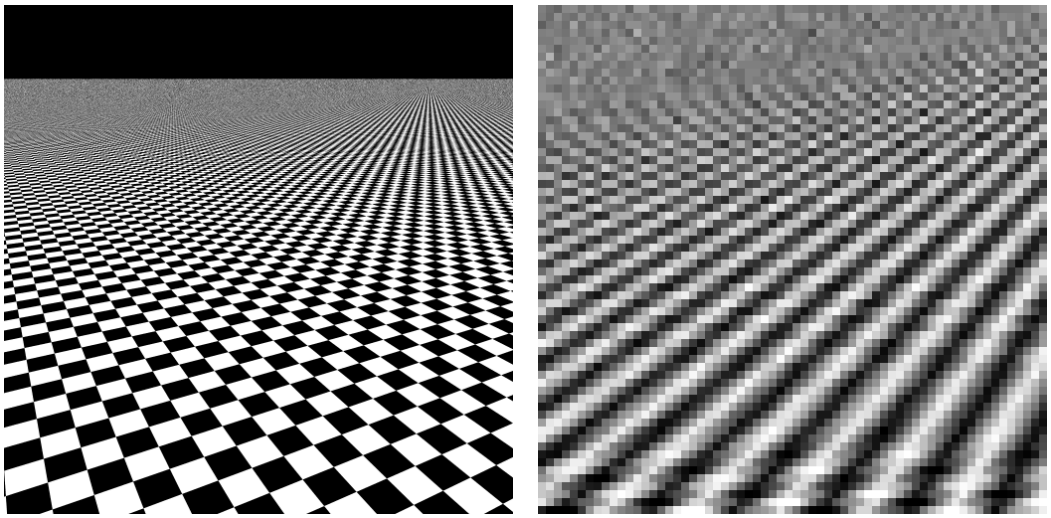
$$f(x,y) = \frac{1}{2}\left(1 + 1600\sin\left(x^2 + y^2\right)\right), \quad x, y \in [0,1). \tag{6.1}$$

The results of the different sampling strategies are shown in Figure 6.3, with the corresponding convergence graph Figure 6.4.

This time, the Larcher-Pillichshammer points only perform best for sample counts up to 256 samples per pixel. Beyond that, the permutation-net construction yields better results. Interestingly, Hammersley-points, rank-1 lattices, and the points generated with optimized matrices partially perform worse than jittered grid sampling.

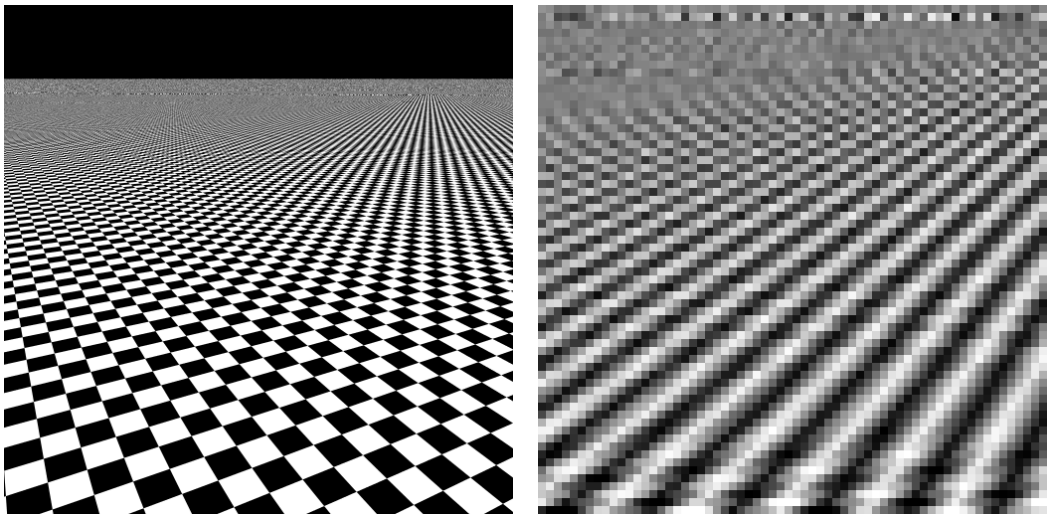


(a) Reference image, with $500^2 = 250\,000$ jittered grid samples per pixel.

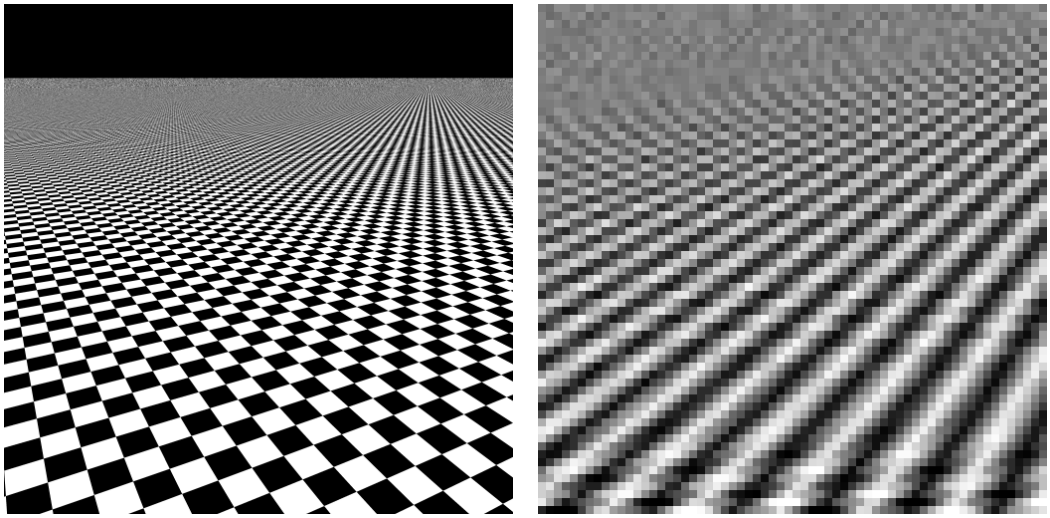


(b) Jittered grid, with 64 samples per pixel.

Figure 6.1: Results for the Checkerboard scene, part 1.

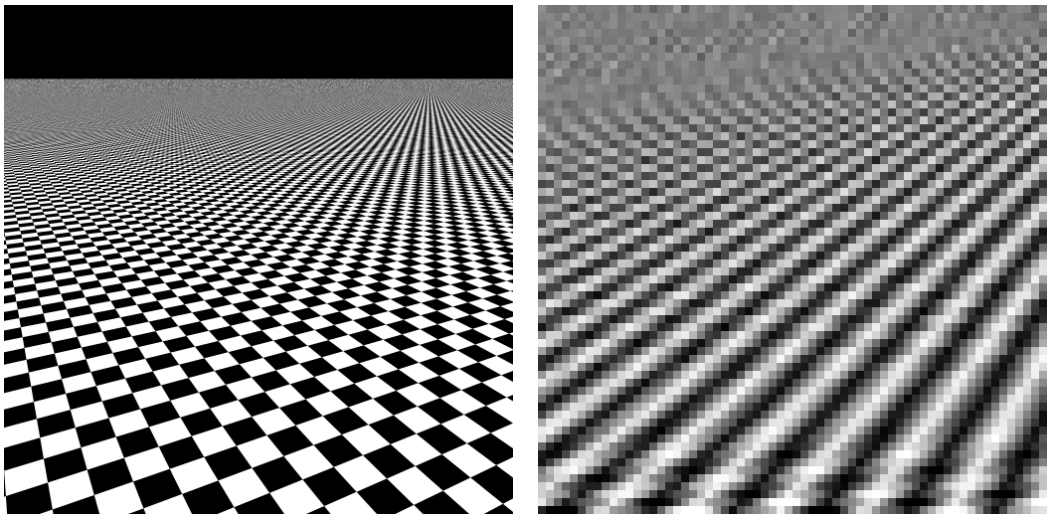


(c) Maximized minimum distance rank-1 lattice, with 64 samples per pixel. Note the visible line at the top of the magnified region. There are still visible artifacts due to correlation with the diagonals of the lattice even though we used a Cranley-Patterson rotation per pixel.

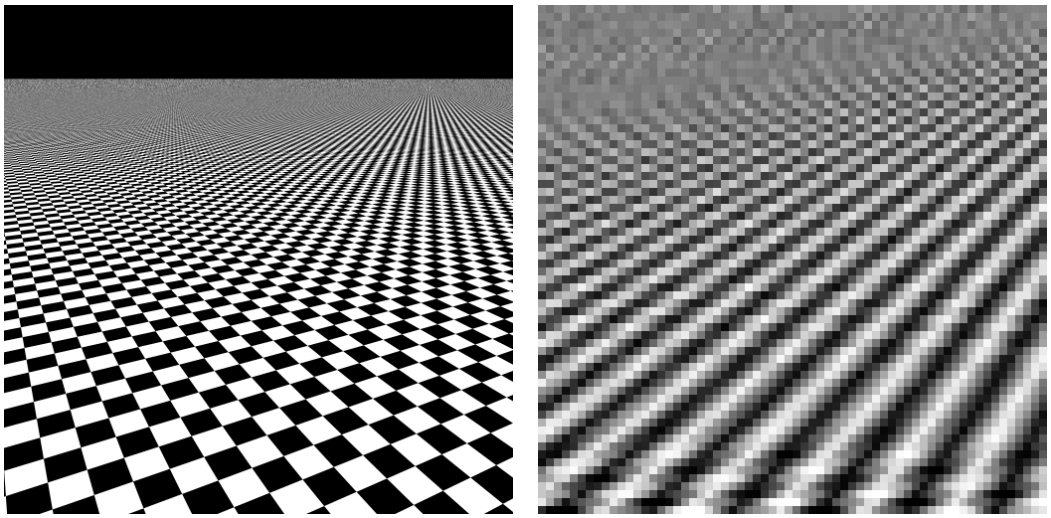


(d) Hammersley net, full-screen sampling, with 64 samples per pixel.

Figure 6.1: Results for the Checkerboard scene, part 2.

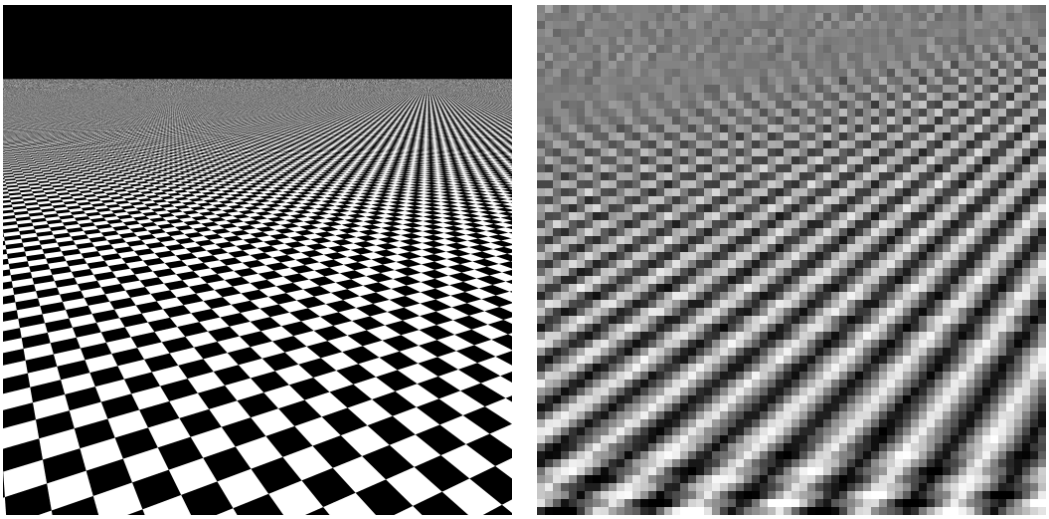


(e) Larcher-Pillichshammer net, full-screen sampling, with 64 samples per pixel.



(f) Optimized matrix-generated net [GHSK07], full-screen sampling, with 64 samples per pixel.

Figure 6.1: Results for the Checkerboard scene, part 3.



(g) Permutation-generated net, full-screen sampling, with 64 samples per pixel.

Figure 6.1: Results for the Checkerboard scene, part 4.

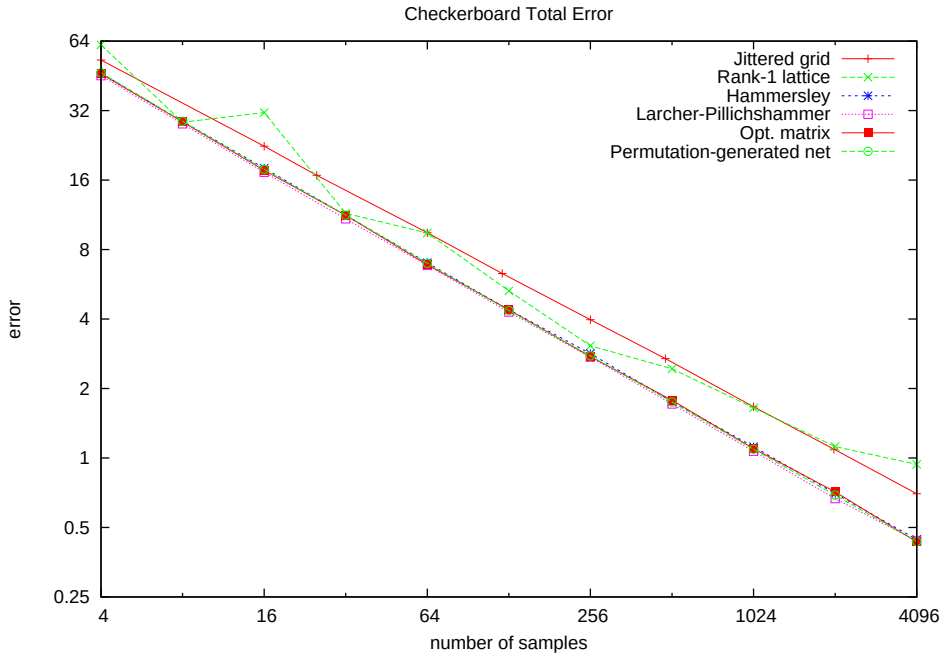
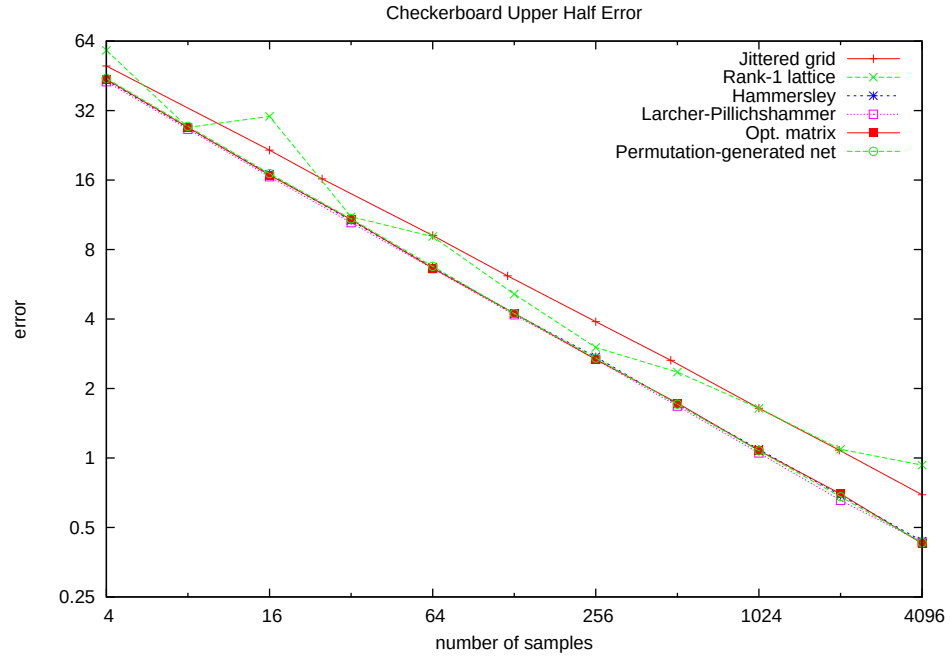
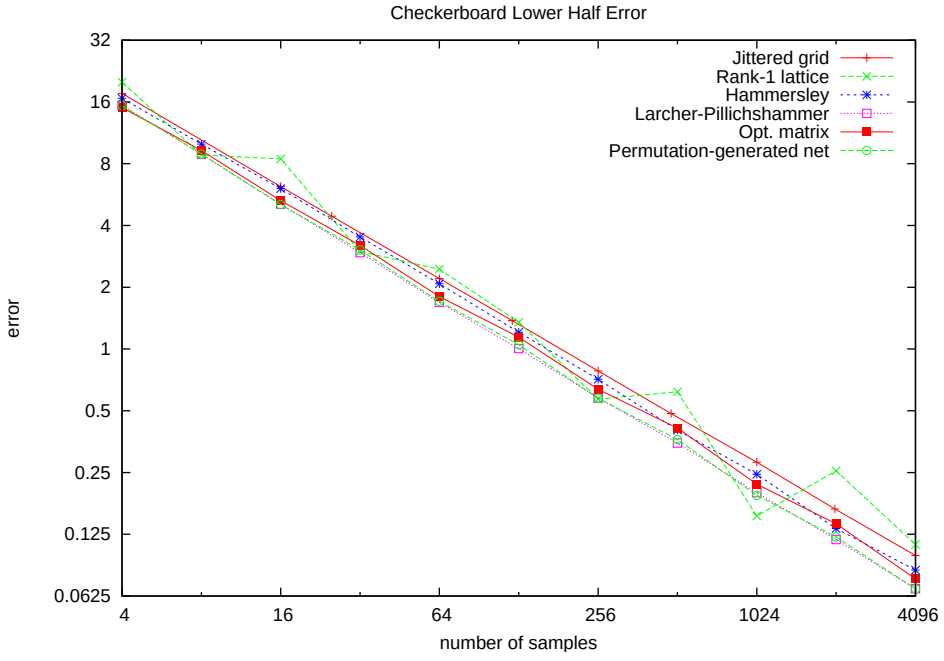


Figure 6.2: This convergence graph depicts the error for the different samplers for increasing sample counts. Please note that the axes are scaled logarithmically. In addition to the total error we have measured the error for the upper and the lower half of the image separately. The Larcher-Pillichshammer net performs best for this scene, while the convergence of the rank-1 lattice fluctuates depending on the sample count.

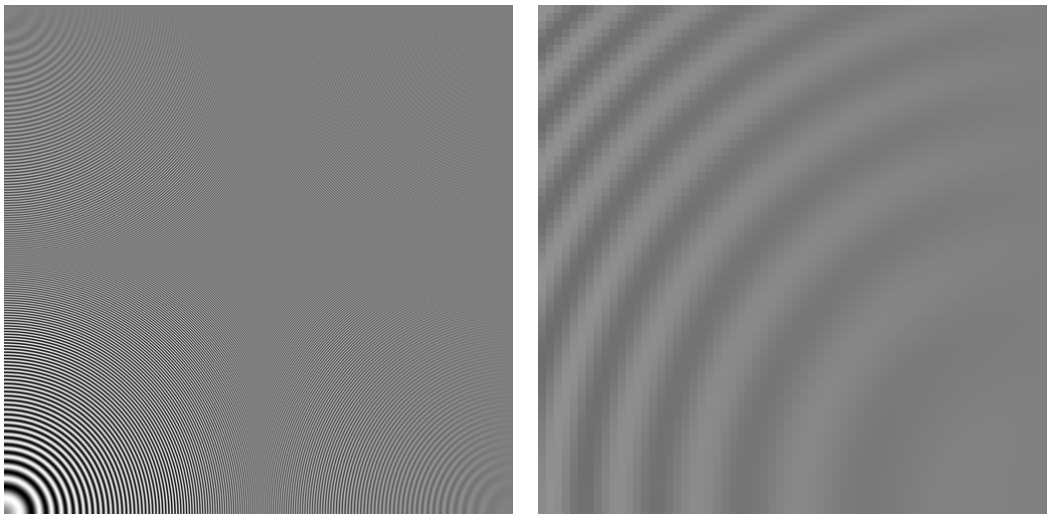


(b)

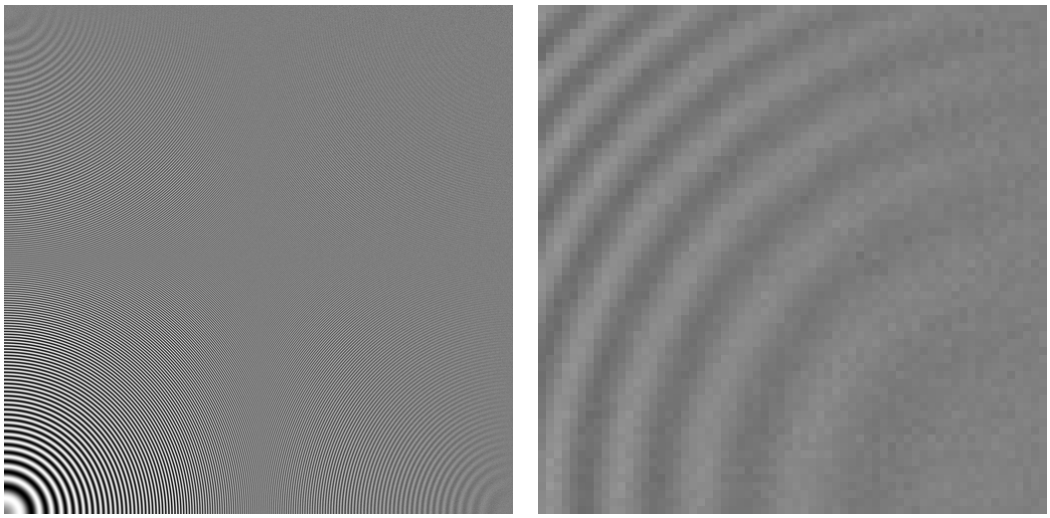


(c)

Figure 6.2: The remaining convergence graphs for the checkerboard scene.

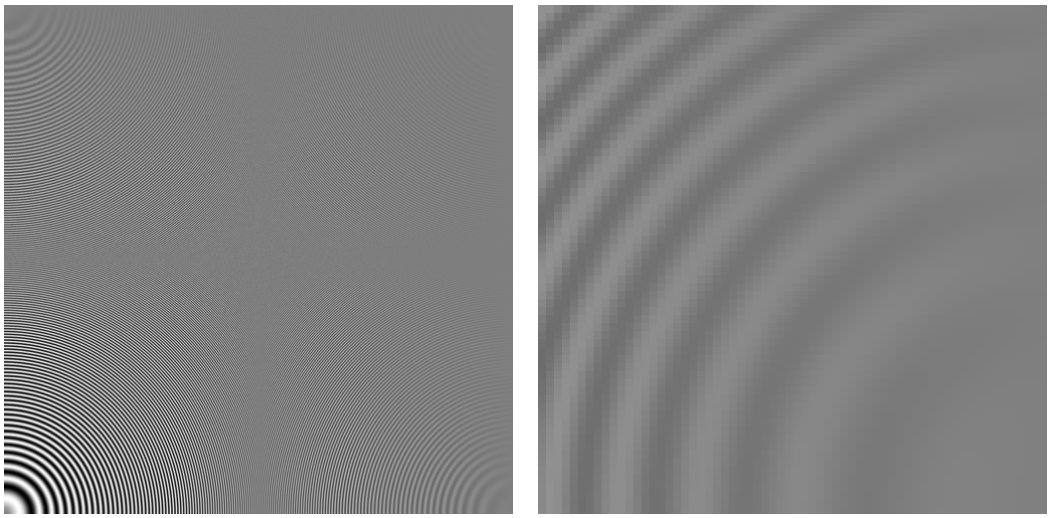


(a) Reference image, with 500^2 = 250 000 jittered grid samples per pixel.

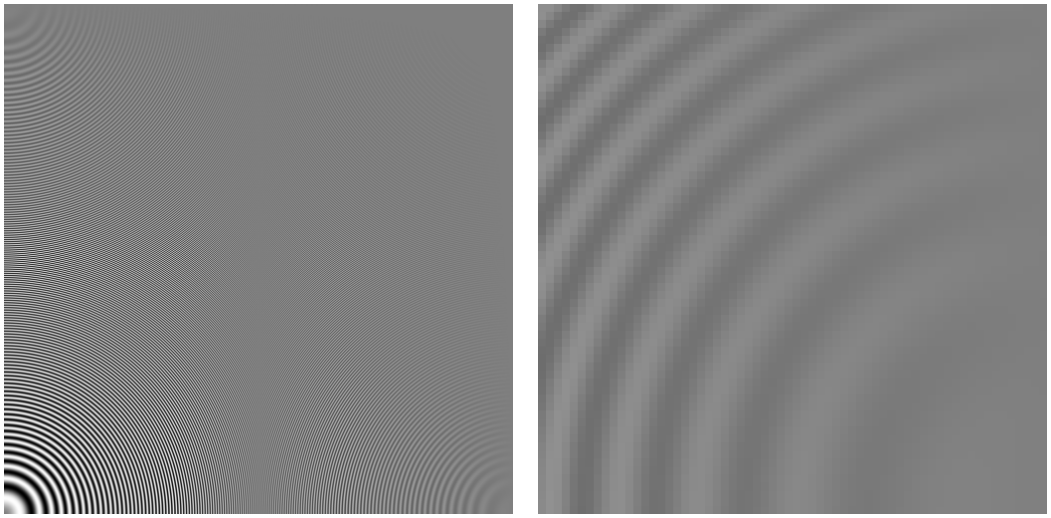


(b) Jittered grid, with 64 samples per pixel.

Figure 6.3: Results for the zone plate test pattern, part 1.

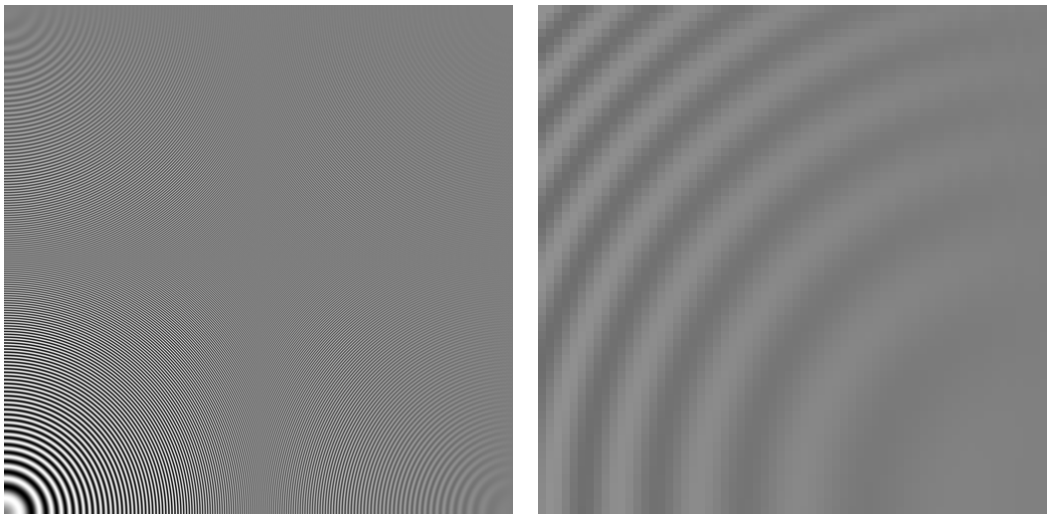


(c) Maximized minimum distance rank-1 lattice, with 64 samples per pixel. Note the visible line at the top of the magnified region. There are still visible artifacts due to correlation with the diagonals of the lattice even though we used a Cranley-Patterson rotation per pixel.

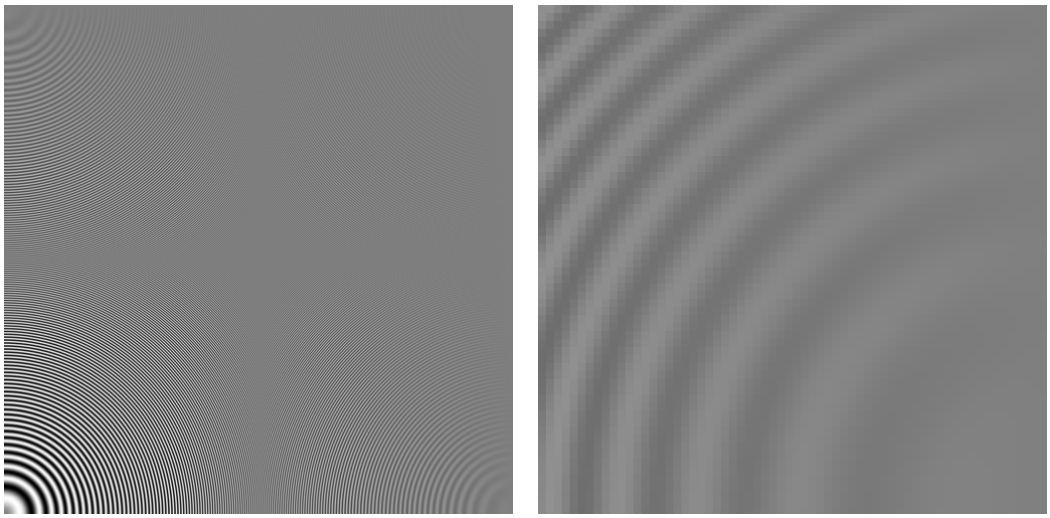


(d) Hammersley net, full-screen sampling, with 64 samples per pixel.

Figure 6.3: Results for the zone plate test pattern, part 2.

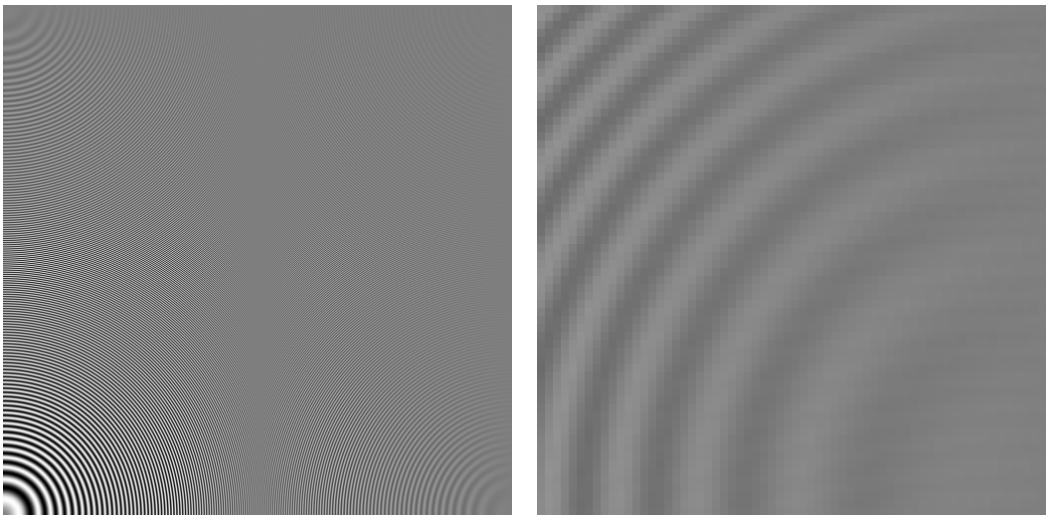


(e) Larcher-Pillichshammer net, full-screen sampling, with 64 samples per pixel.



(f) Optimized matrix-generated net [GHSK07], full-screen sampling, with 64 samples per pixel.

Figure 6.3: Results for the zone plate test pattern, part 3.



(g) Permutation-generated net, full-screen sampling, with 64 samples per pixel.

Figure 6.3: Results for the zone plate test pattern, part 4.

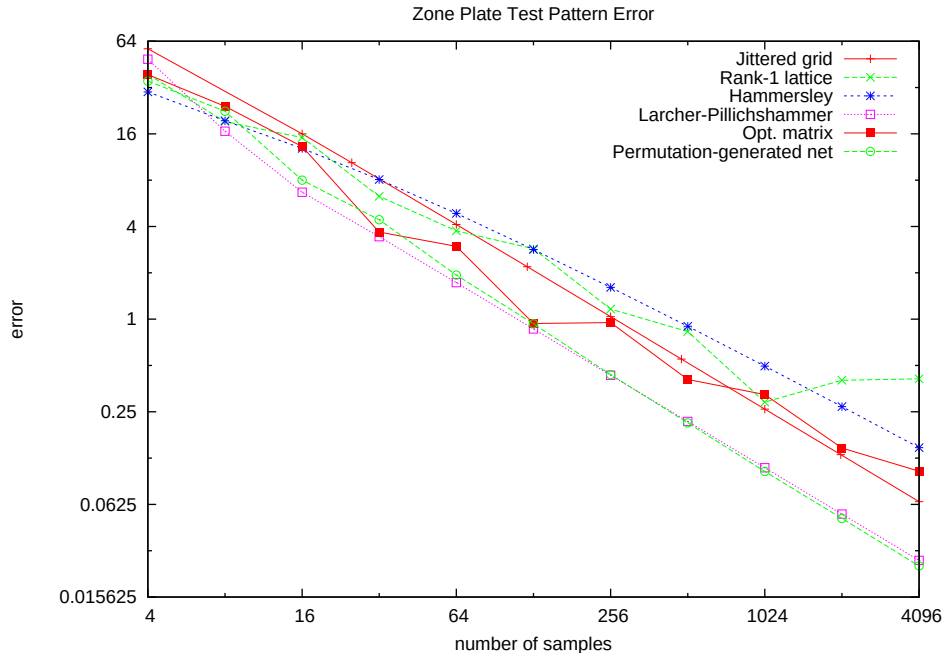


Figure 6.4: This convergence graph depicts the error for the different samplers for increasing sample counts. Please note that the axes are scaled logarithmically. Interestingly, the Hammersley point sets perform worse than jittered sampling. This time, the permutation-generated nets perform best for large sample counts.

6.2 Sampling in Dimension $s = 3$

As we have seen in the chapters above it makes sense to use both ray tracing and numerical sampling methods to simulate motion blur. For the following tests we have used 3D-sampling points which fit perfectly into the Reyes pipeline when depth of field effects are not considered.

To render these scenes we have used a BVH built over the whole animation with the SAH. We would like to note that the results are completely independent of the acceleration structure that is used, though.

We tried two different scenes, namely a Cornell box and a helicopter scene. Both scenes feature difficult motion blur settings, described in more detail below. Only the direct lighting of point light sources was simulated to emphasize the specific motion blur artifacts. The following samplers have been used to generate points in $[0,1)^3$ (two components were used for anti-aliasing, the remaining component was used for temporal stratification):

- Uniform random samples. This corresponds to the pure Monte Carlo method.
- Jittered grid. This reduces variance because of the better stratification.
- Blue noise samples, generated using Lloyd relaxation [HDK01].
- Rank-1 lattices. We used generator vectors that yield lattices with very large (often optimal) minimum distance [DKD07].
- Sobol' $(0, m, 3)$-nets.
- Modified $(0, m, 3)$-nets. Here one of the two-dimensional projections yields that Larcher-Pillichshamer $(0, m, 2)$-net. We used this projection for anti-aliasing.

For all deterministic samplers above, we used a Cranley-Patterson rotation to randomize the sample set for each pixel. Otherwise the results are visually inacceptable due to the correlation for small numbers of samples.

In addition to per-pixel sample sets we also used $(0, m, 3)$-nets that are stratified over the whole screen. This way, a Cranley-Patterson rotation is not needed anymore. While this is definitely a very nice feature, new artifacts inherent to the $(0, m, 3)$-net structure may appear. The techniques of Subsection 5.2.4 can be used to parallelize the rendering efficiently when using such full-screen nets. We used both the Sobol'- and the modified Sobol'-net for full-screen sampling in our numerical tests.

The following figures show images that have been computed using 64 samples per pixel (except for the reference images with 125 000 samples per pixel). The image on the right of each figure depicts a magnified area of the image. A convergence plot for different sample counts can be found at the end of each section for both test scenes.

6.2.1 Cornell Box

This scenes features a moving reflective sphere and a ball that jumps on a discontinuous trajectory. A moving highlight is visible on the ball and the shadow cast by the ball is very

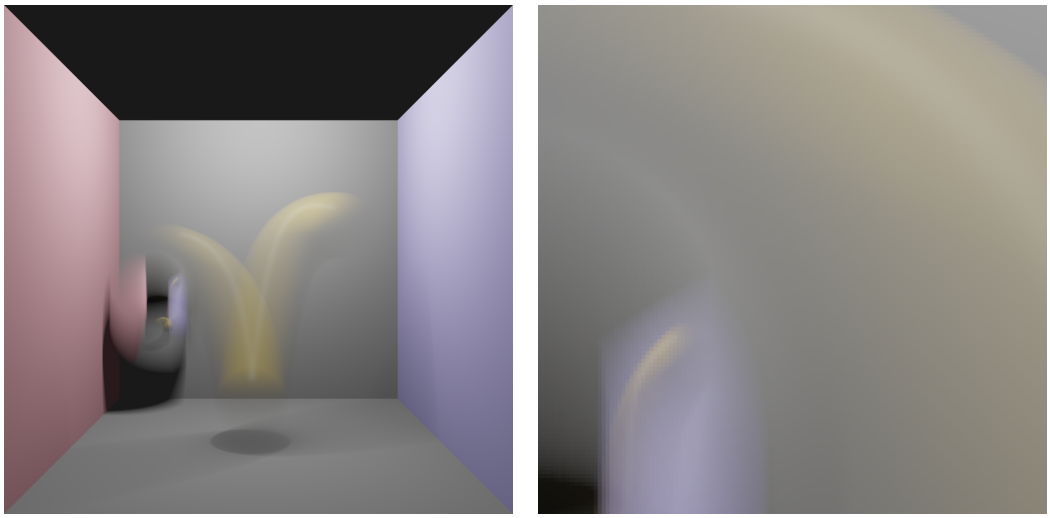
complex even though only the direct lighting from point light sources was simulated. The results of the different sampling strategies are shown in Figure 6.5, with the corresponding convergence graph in Figure 6.6.

The Sobol'-net and the modified Sobol'-net perform almost equally well in this scene, outperforming the other samplers for almost all sample counts. However the full-screen nets perform worse than those with a Cranley-Patterson rotation applied per pixel.

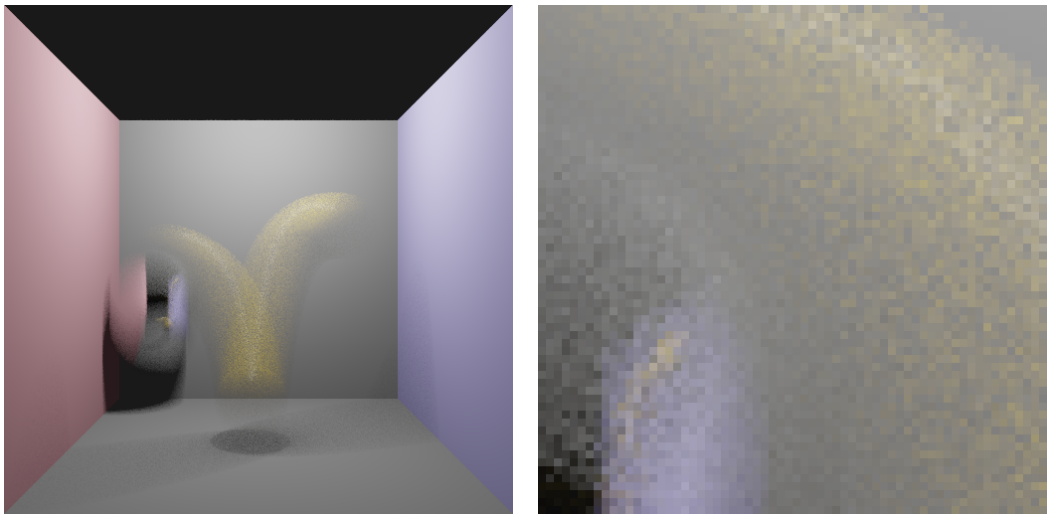
6.2.2 KA-58 Helicopter

This scenes features a flying helicopter with moving rotors. The ground itself does not move, but its checkerboard shading pattern moves non-linearly. This pattern is especially difficult for rank-1 lattices due to correlations between the hyperplanes of the lattice and the diagonals apparent in the shading structure. The results of the different sampling strategies are shown in Figure 6.7, with the corresponding convergence graph in Figure 6.8.

Again, the Sobol'-nets perform best, yet the full-screen variants yield the best results for this scene. This could be related to the checkerboard ground pattern, also resulting in correlations with the hyperplanes of the rank-1 lattice.

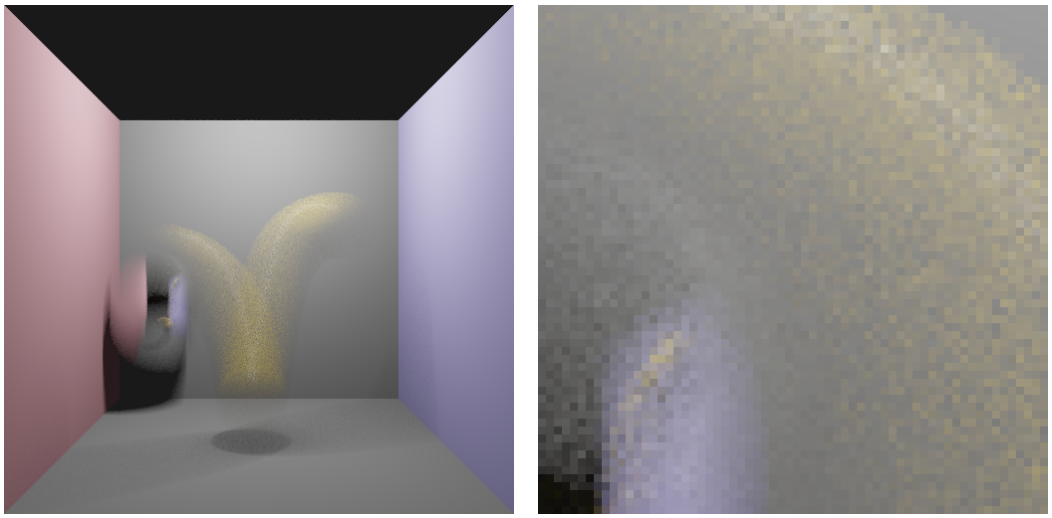


(a) Reference image, with $50^3 = 125\,000$ jittered grid samples per pixel.

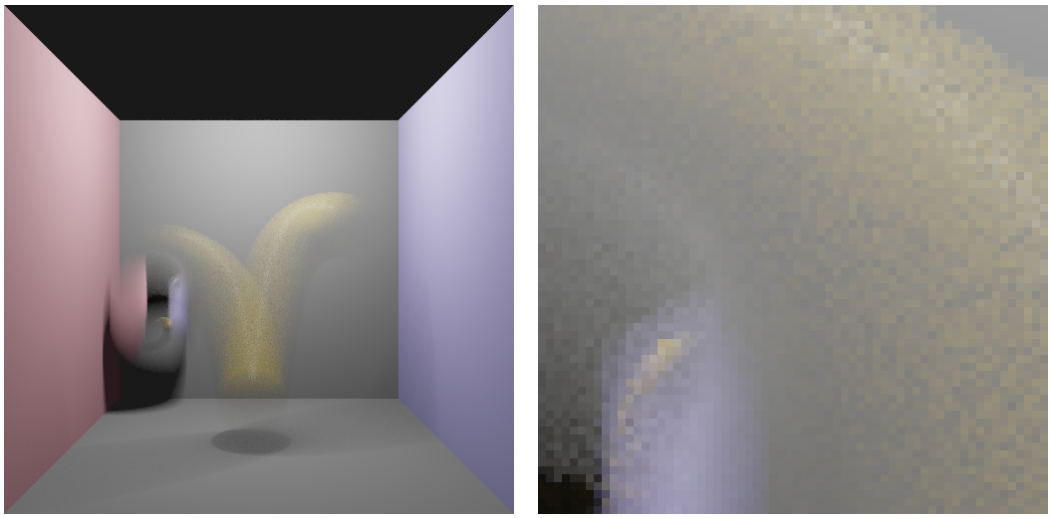


(b) Random, with 64 samples per pixel.

Figure 6.5: Results for the Cornell box scene, part 1.

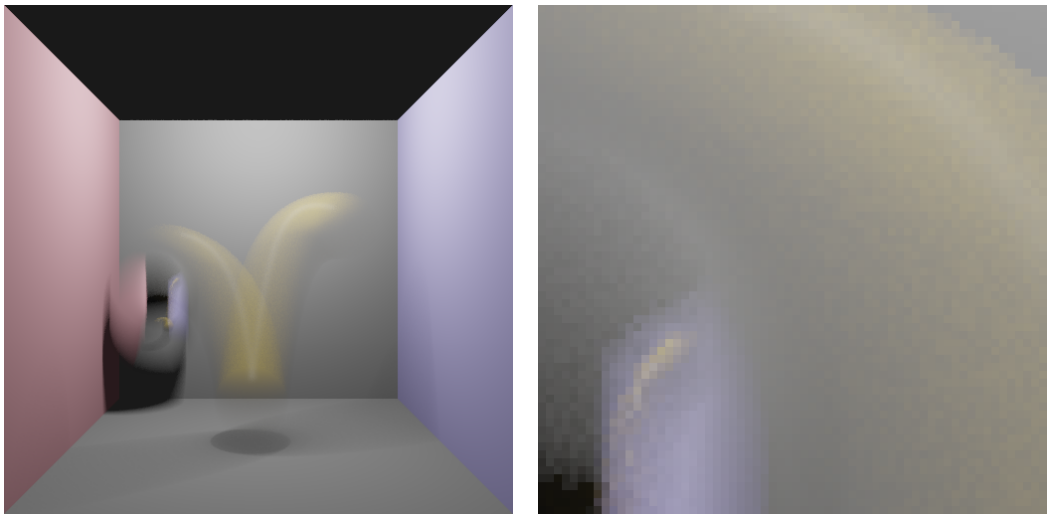


(c) Jittered grid, with 64 samples per pixel.

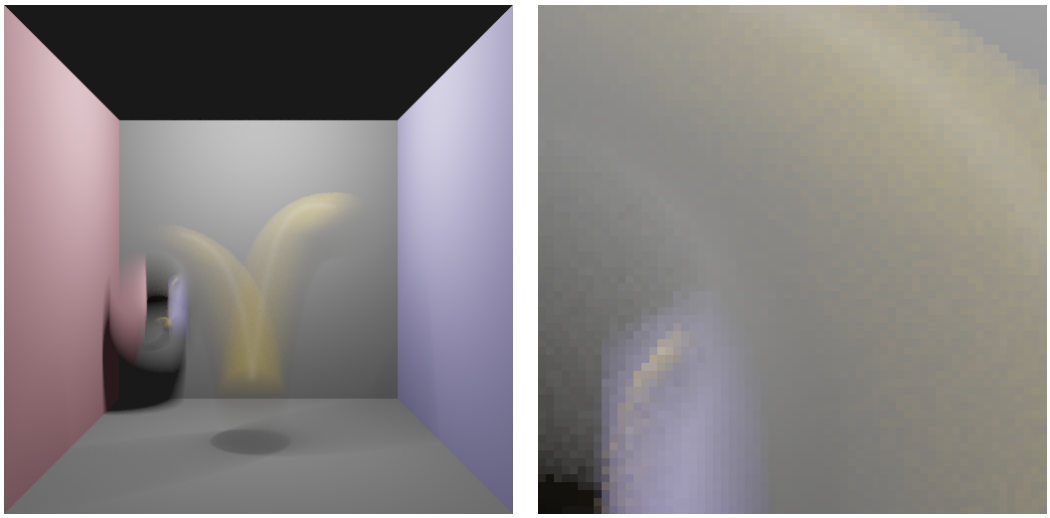


(d) Lloyd relaxation, with 64 samples per pixel.

Figure 6.5: Results for the Cornell box scene, part 2.

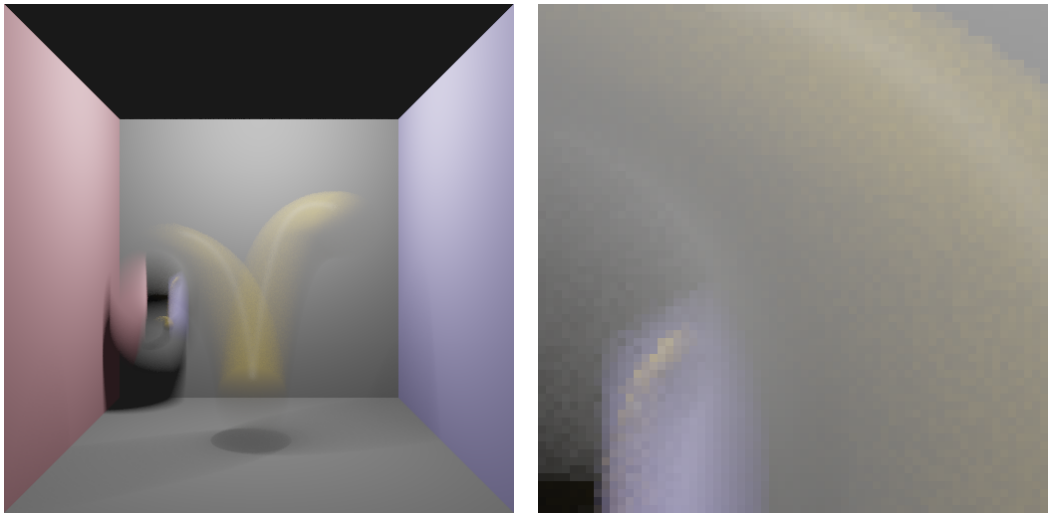


(e) Maximized minimum distance rank-1 lattice, with 64 samples per pixel.

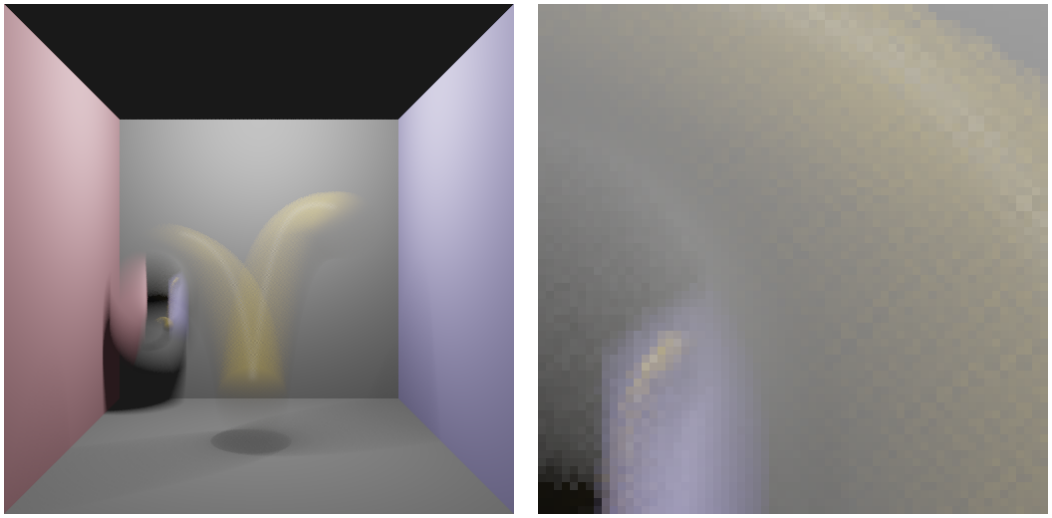


(f) Sobol' $(0, 6, 3)$-net, with 64 samples per pixel.

Figure 6.5: Results for the Cornell box scene, part 3.

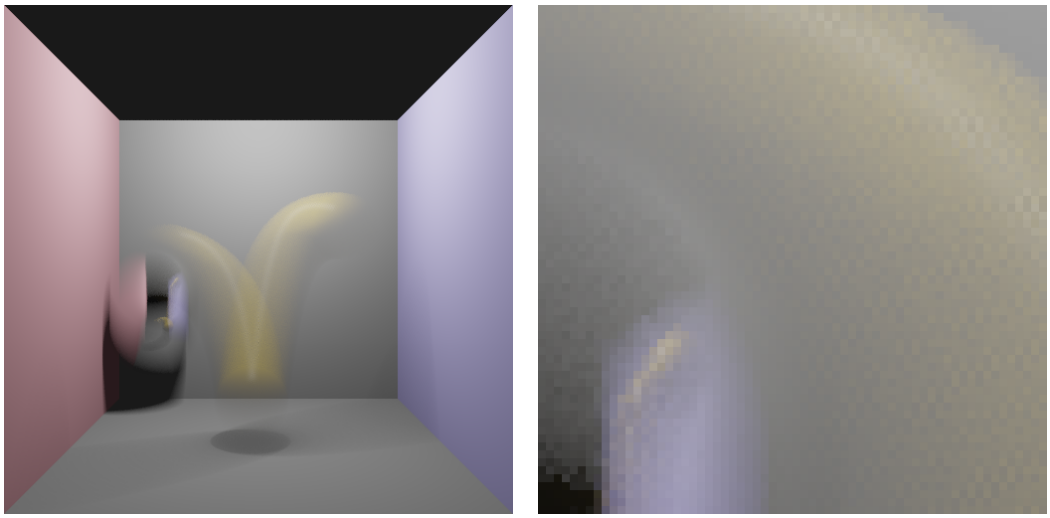


(g) Modified Sobol' $(0, 6, 3)$-net, with 64 samples per pixel. The two dimensional projection used for anti-aliasing is equal to the Larcher-Pillichshammer $(0, 6, 2)$-net.



(h) Sobol' $(0, m, 3)$-net, full-screen sampling, with 64 samples per pixel.

Figure 6.5: Results for the Cornell box scene, part 4.



(i) Modified Sobol' $(0, m, 3)$-net, full-screen sampling, with 64 samples per pixel. The two dimensional projection used for anti-aliasing is equal to the Larcher-Pillichshammer $(0, m, 2)$-net.

Figure 6.5: Results for the Cornell box scene, part 5.

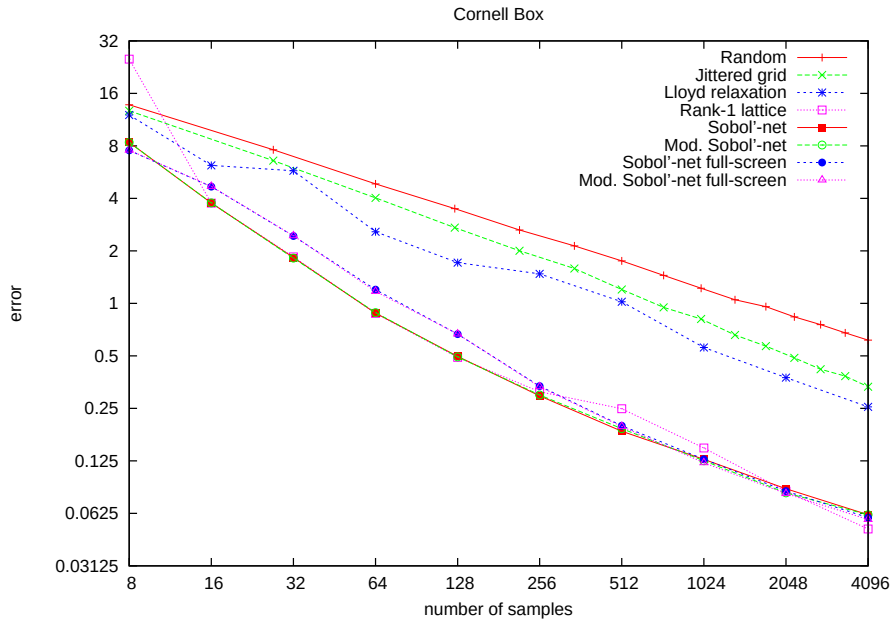
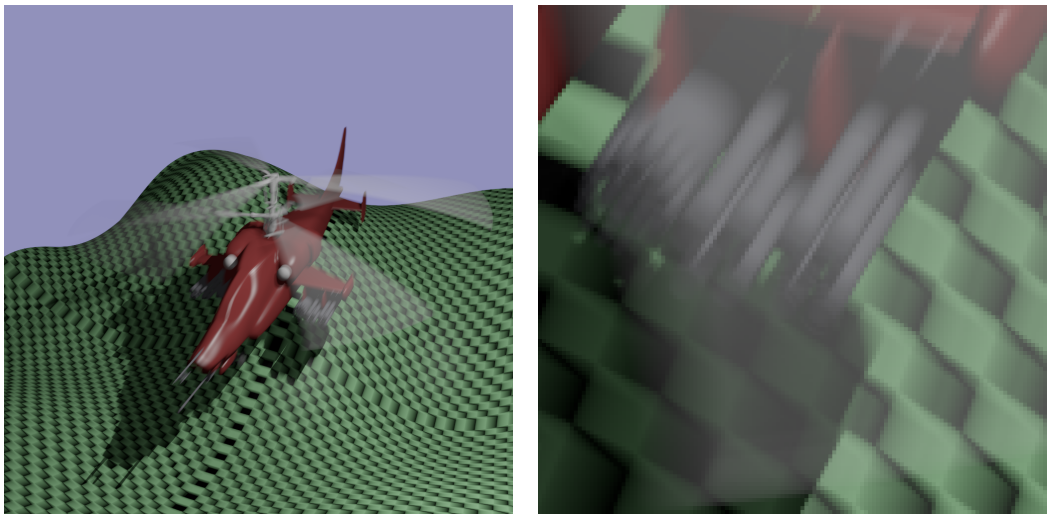
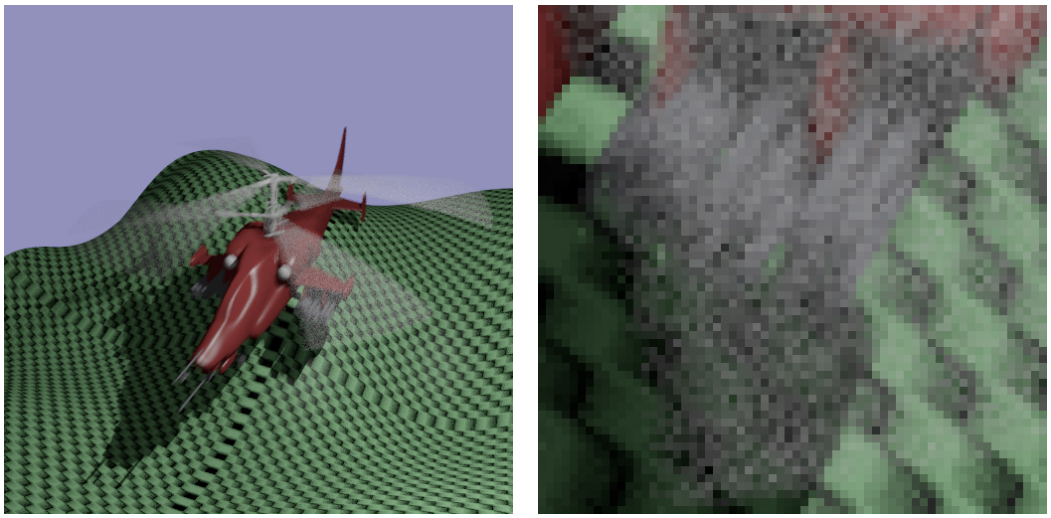


Figure 6.6: This convergence graph depicts the error for the different samplers for increasing sample counts. Please note that the axes are scaled logarithmically. The modified Sobol'-nets perform only marginally better than the original Sobol'-net in this scene. The rank-1 lattice even outperforms these nets sometimes, but the error does not decrease as smoothly. Surprisingly, for this scene the full-screen nets perform worse than the randomized Sobol'-nets. Especially for small sample counts the artifacts caused by the structure of the $(0, m, 3)$-net are visually apparent (see Figure 6.5 (h) and (i)).

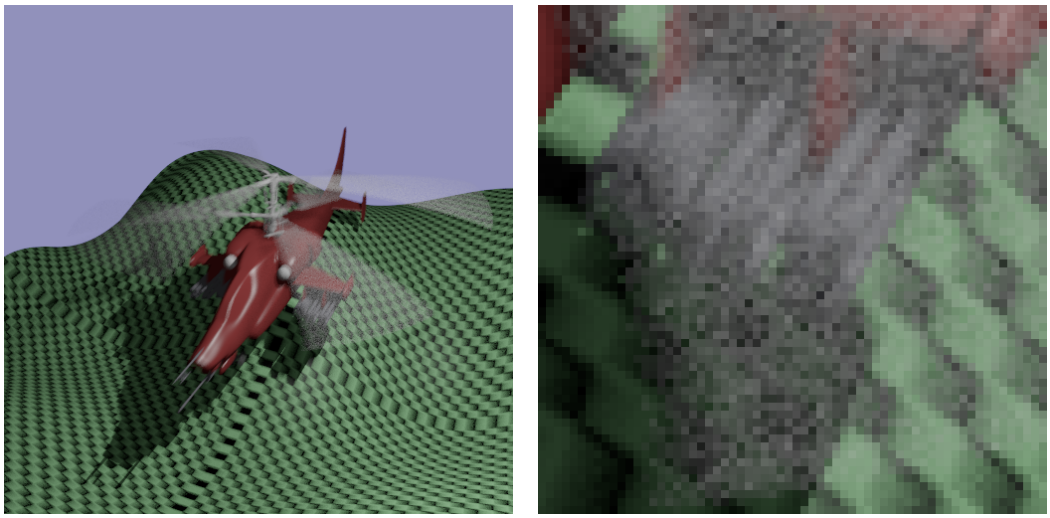


(a) Reference image, with $50^3 = 125\,000$ jittered grid samples per pixel.

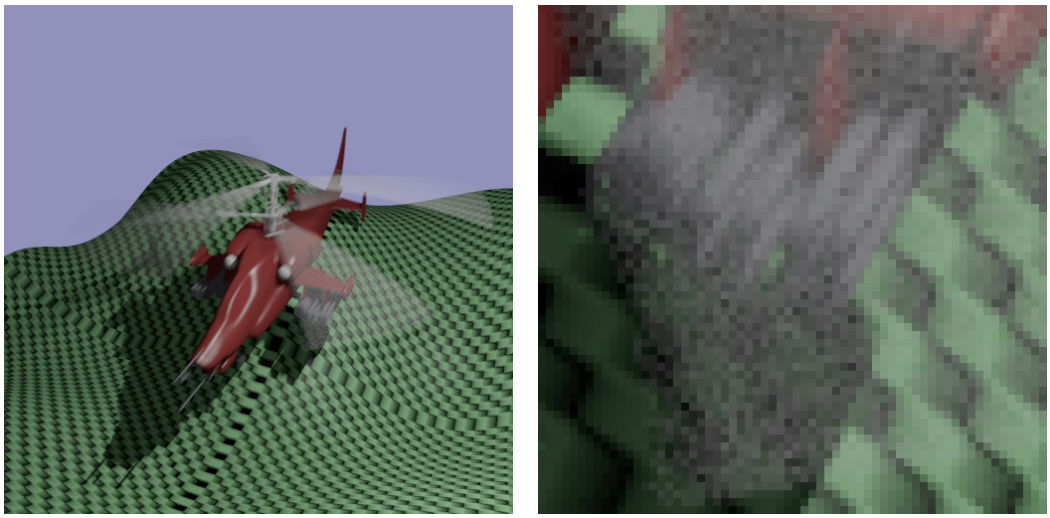


(b) Random, with 64 samples per pixel.

Figure 6.7: Results for the KA-58 helicopter scene, part 1.

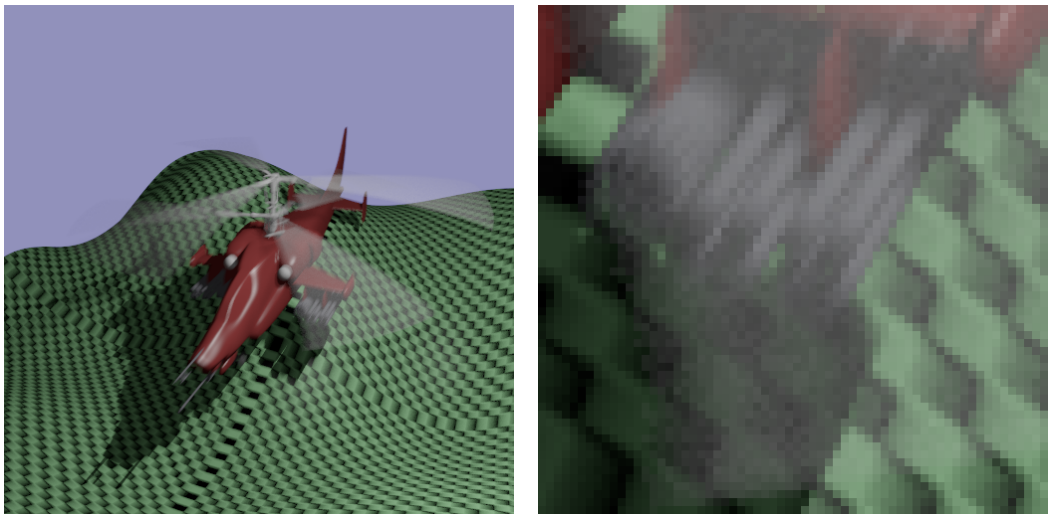


(c) Jittered grid, with 64 samples per pixel.

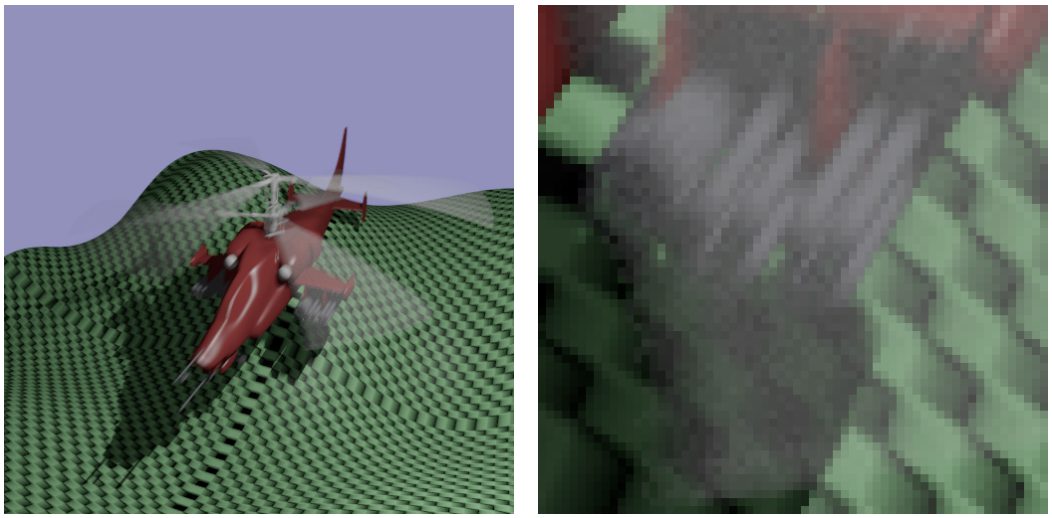


(d) Lloyd relaxation, with 64 samples per pixel.

Figure 6.7: Results for the KA-58 helicopter scene, part 2.

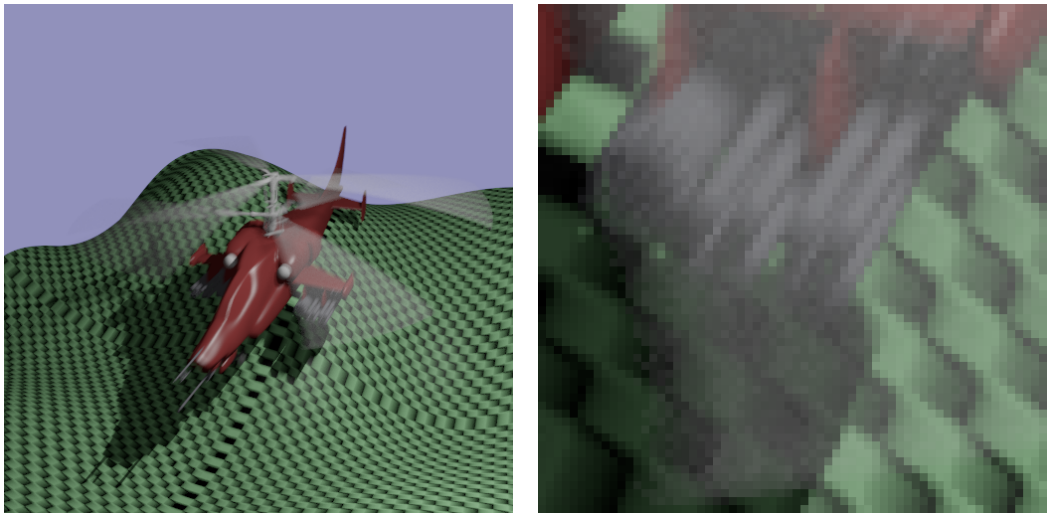


(e) Maximized minimum distance rank-1 lattice, with 64 samples per pixel.

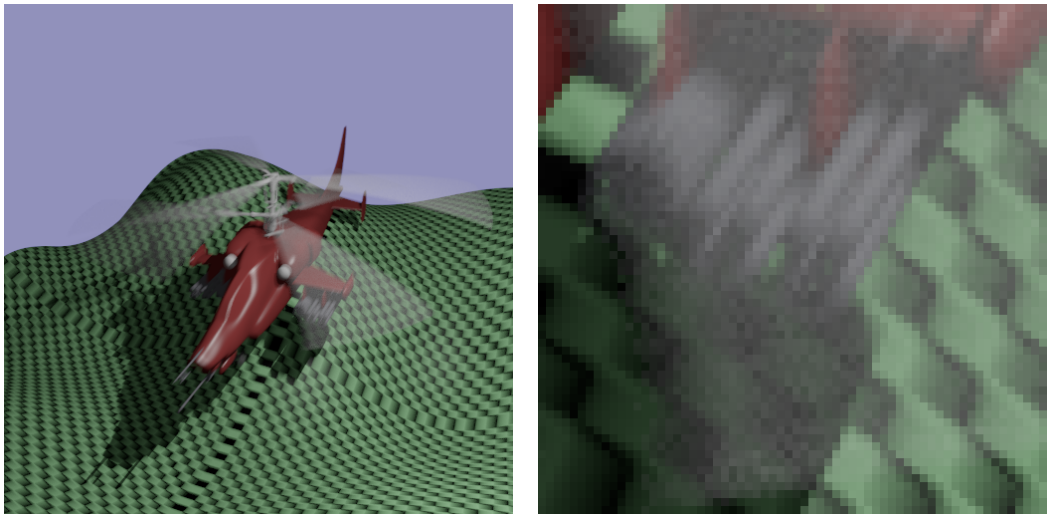


(f) Sobol' $(0, 6, 3)$-net, with 64 samples per pixel.

Figure 6.7: Results for the KA-58 helicopter scene, part 3.

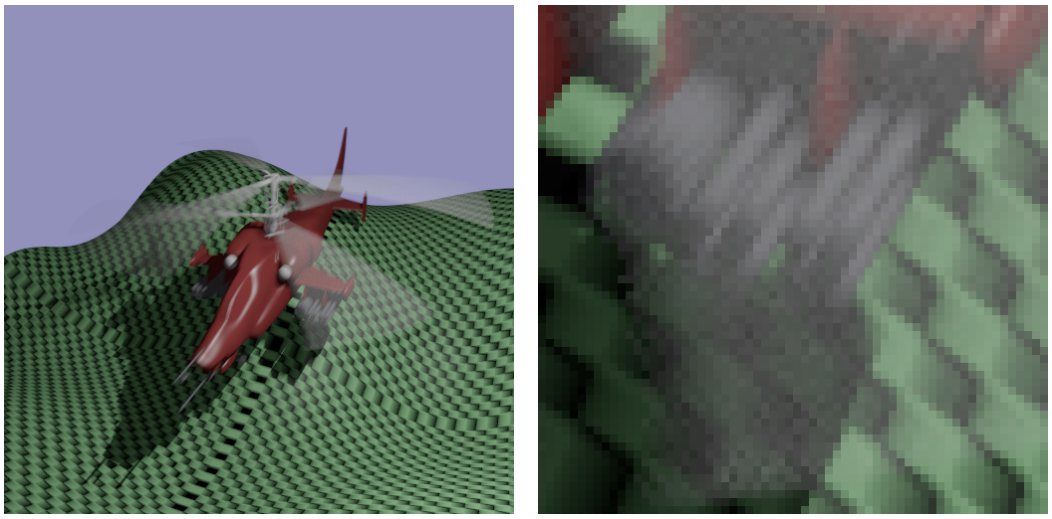


(g) Modified Sobol' $(0, 6, 3)$-net, with 64 samples per pixel. The two dimensional projection used for anti-aliasing is equal to the Larcher-Pillichshammer $(0, 6, 2)$-net.



(h) Sobol' $(0, m, 3)$-net, full-screen sampling, with 64 samples per pixel.

Figure 6.7: Results for the KA-58 helicopter scene, part 4.



(i) Modified Sobol' $(0, m, 3)$-net, full-screen sampling, with 64 samples per pixel. The two dimensional projection used for anti-aliasing is equal to the Larcher-Pillichshammer $(0, m, 2)$-net.

Figure 6.7: Results for the KA-58 helicopter scene, part 5.

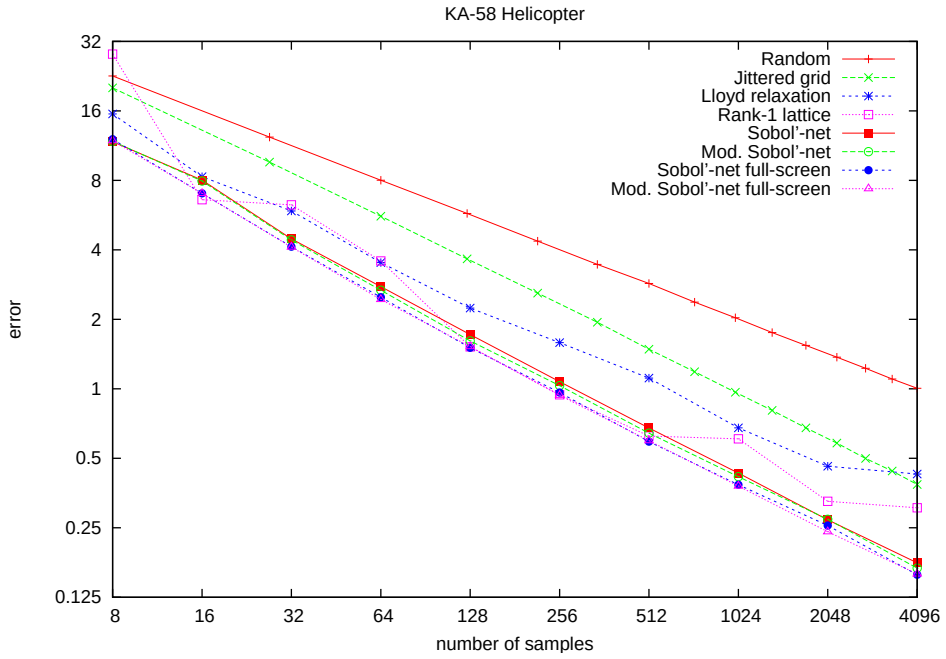


Figure 6.8: This convergence graph depicts the error for the different samplers for increasing sample counts. Please note that the axes are scaled logarithmically. Apparently the rank-1 lattice sampler has problems capturing the checkerboard ground plane due to correlations with the hyperplanes of the lattice. This time the full-screen Sobol'-nets perform significantly better, with slightly better results using the modified Sobol'-net.

7 Summary

Motion blur is a mathematically hard problem and only very few algorithms are sufficiently robust to handle complex scenes. Partly that is due to the underlying radiance function that needs to be integrated. This function can be quite arbitrary, i.e. it usually not only varies rapidly in both space and time, but is unpredictably discontinuous.

A multitude of algorithms has been developed to render motion blur. Besides variations of the Monte Carlo approach a promising analytic method has been described that reduces variance while making ray intersections more expensive.

All production renderers at least partially employ a ray tracing framework, where an acceleration structure needs to be carefully tuned to achieve fast performance. The currently best known method to build these structures is the SAH. This local heuristic can be extended to handle the time dimension in order to efficiently process rays distributed in time. Interpreting the SAH as a mean to minimize the sum of the bounding box areas of the acceleration structure somewhat explains why the SAH works so well in practice.

While some parts of the rendering process may be computed analytically, at some point one needs to resort to numerical methods based on sampling. While the original Monte Carlo method uses random samples to estimate integrals, typically convergence is greatly improved by using very well stratified quasi-Monte Carlo points. New constructions for both two- and three-dimensional points with very large minimum distance have been described in this thesis.

Ray tracing can be combined perfectly with these newly developed quasi-Monte Carlo samplers. A comparison of different kinds of sample points confirmed that quasi-Monte Carlo points are indeed superior to their random counterparts. However the choice of the best sampling points seems to be drastically influenced by the scene (i.e. the function that is integrated). It is therefore hard to say which points are "best" – this also depends on the number of samples used and is hard to predict.

Optimizing point sets by enlarging the minimum distance yielded good results. Nevertheless previous constructions are not always inferior. Future work includes looking for both higher-dimensional point sets and other measures to optimize, e.g. discrepancy and dispersion.

Bibliography

[AGP+95] T. Apodaca, L. Gritz, T. Porter, O. Jacob, M. Turner, J. Letteri, E. Poon, and H. Zargarpour. Using RenderMan in animation production. In *SIGGRAPH '95: ACM SIGGRAPH 1995 Courses*. ACM Press, 1995. (Cited on pages 2, 24, and 29.)

[AK87] J. Arvo and D. Kirk. Fast ray tracing by ray classification. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 55–64, New York, NY, USA, 1987. ACM. (Cited on page 51.)

[Ale02] M. Alexa. Linear combination of transformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 380–387, New York, NY, USA, 2002. ACM Press. Also see [BBM04]. (Cited on page 6.)

[AMMH07] T. Akenine-Möller, J. Munkberg, and J. Hasselgren. Stochastic rasterization using time-continuous triangles. In *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 7–16, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. (Cited on pages 18 and 35.)

[AV07] D. Arthur and S. Vassilvitskii. `k-means++`: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics. (Cited on page 45.)

[BBM04] C. Bloom, J. Blow, and C. Muratori. Errors and omissions in Marc Alexa's "Linear combinations of transformations", 2004. Available at `http://www.cbloom.com/3d/techdocs/lcot_errors.pdf`. (Cited on pages 6 and 125.)

[BCGH92] A. Barr, B. Currin, S. Gabriel, and J. Hughes. Smooth interpolation of orientations with angular velocity constraints using quaternions. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 313–320, New York, NY, USA, 1992. ACM Press. (Cited on page 10.)

[BE01] G. Brostow and I. Essa. Image-based motion blur for stop motion animation. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 561–566, New York, NY, USA, 2001. ACM Press. (Cited on pages 29 and 34.)

[Bez01] A. Bezerianos. Using projection to accelerate ray tracing. Master's thesis, University of Toronto, October 2001. (Cited on page 39.)

[BF88] P. Bratley and B. Fox. Algorithm 659: implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14(1):88–100, 1988. (Cited on page 58.)

[BFMZ94] G. Bishop, H. Fuchs, L. McMillan, and E. Scher Zagier. Frameless rendering: double buffering considered harmful. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 175–176, New York, NY, USA, 1994. ACM. (Cited on page 22.)

[BH07] J. Bittner and V. Havran. RDH: Ray distribution heuristics for construction of spatial data structures. In *Proceedings of Symposium on Interactive Ray Tracing 2007 (posters)*, 2007. (Cited on page 39.)

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In H. Garcia-Molina and H. Jagadish, editors, *SIGMOD Conference*, pages 322–331. ACM Press, 1990. (Cited on pages 39 and 44.)

[Blo99] M. Bloomenthal. Error bounded approximate reparametrization of non-uniform rational B-spline curves, 1999. (Cited on page 8.)

[BPT02] S. Brakatsoulas, D. Pfoser, and Y. Theodoridis. Revisiting R-tree construction principles. In *ADBIS '02: Proceedings of the 6th East European Conference on Advances in Databases and Information Systems*, pages 149–162, London, UK, 2002. Springer-Verlag. (Cited on pages 45, 46, and 47.)

[But02] B. Butterworth. The twelve days of christmas: Music meets math in a popular christmas song. Technical report, Inside Science News Service, 2002. Available at `http://www.aip.org/isns/reports/2002/058.html`. (Cited on page 63.)

[Cat84] E. Catmull. An analytic visible surface algorithm for independent pixel processing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 109–115, New York, NY, USA, 1984. ACM Press. (Cited on pages 27 and 28.)

[CCC87] R. Cook, L. Carpenter, and E. Catmull. The Reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, 1987. (Cited on page 29.)

[CD05] E. Chan and F. Durand. Fast prefiltered lines. In R. Fernando, editor, *GPU Gems 2*, chapter 22, pages 345–359. Addison-Wesley, March 2005. (Cited on page 28.)

[CFLB06] P. Christensen, J. Fong, D. Laur, and D. Batali. Ray tracing for the movie 'Cars'. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6, September 2006. (Cited on page 48.)

[CJ02] M. Cammarano and H. Jensen. Time dependent photon mapping. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 135–144, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association. (Cited on page 19.)

[Coo86] R. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, 1986. (Cited on page 54.)

[CPC84] R. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, New York, NY, USA, 1984. ACM Press. (Cited on pages 2, 17, 19, and 29.)

[Dam05] S. Dammertz. Image synthesis by rank-1 lattices. Master's thesis, University of Ulm, 2005. (Cited on pages 71 and 88.)

[DBBS06] P. Dutre, K. Bala, P. Bekaert, and P. Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006. (Cited on page 21.)

[Dem04] J. Demers. Depth of field: A survey of techniques. In R. Fernando, editor, *GPU Gems*, chapter 23, pages 375–390. Addison-Wesley, March 2004. (Cited on page 35.)

[DK07] S. Dammertz and A. Keller. Image synthesis by rank-1 lattices. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 217–236. Springer, 2007. (Cited on pages 54, 62, and 82.)

[DKD07] H. Dammertz, A. Keller, and S. Dammertz. Simulation on rank-1 lattices. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 205–216. Springer, 2007. (Cited on pages 92, 99, and 110.)

[DL04] O. Deussen and B. Lintermann. *Digital Design of Nature: Computer Generated Plants and Organics*. Springer-Verlag, 2004. (Cited on page 54.)

[DSDD07] C. Dachsbacher, M. Stamminger, G. Drettakis, and F. Durand. Implicit visibility and antiradiance for interactive global illumination. *ACM Trans. Graph.*, 26(3):61, 2007. (Cited on page 45.)

[Fau82] H. Faure. Discrépance de suites associées à un système de numération (en dimension *s*). *Acta Arith.*, 41(4):337–351, 1982. (Cited on pages 56, 63, and 67.)

[FK02] I. Friedel and A. Keller. Fast generation of randomized low-discrepancy point sets. In H. Niederreiter, K. Fang, and F. Hickernell, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 257–273. Springer, 2002. (Cited on page 94.)

[FS91] R. Farouki and T. Sakkalis. Real rational curves are not "unit speed". *Comput. Aided Geom. Des.*, 8(2):151–157, 1991. (Cited on page 8.)

[GH96] L. Gritz and J. Hahn. BMRT: a global illumination implementation of the RenderMan standard. *J. Graph. Tools*, 1(3):29–48, 1996. (Cited on page 29.)

[GHSK07] L. Grünschloß, J. Hanika, R. Schwede, and A. Keller. (t, m, s)-nets and maximized minimum distance. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 397–412. Springer, 2007. (Cited on pages 54, 64, 81, 91, 93, 99, 103, and 108.)

[Gla88] A. Glassner. Spacetime ray tracing for animation. *IEEE Comput. Graph. Appl.*, 8(2):60–70, 1988. (Cited on page 50.)

[Gla99] A. Glassner. An open and shut case. *IEEE Comput. Graph. Appl.*, 19(3):82–92, 1999. (Cited on page 17.)

[GM04] X. Guan and K. Mueller. Point-based surface rendering with motion blur. In *Eurographics Symposium on Point Based Graphics*, pages 33–40, 2004. (Cited on page 28.)

[Got00] S. Gottschalk. *Collision queries using oriented bounding boxes*. PhD thesis, The University of North Carolina at Chapel Hill, 2000. Director-Dinesh Manocha and Director-Ming C. Lin. (Cited on page 46.)

[GP90a] E. Gröller and W. Purgathofer. Using temporal and spatial coherence for accelerating the calculation of animation sequences. In *Proceedings of Eurographics '91*, pages 103–113. Elsevier Science Publishers, September 1990. (Cited on page 51.)

[GP90b] B. Guenter and R. Parent. Motion control: Computing the arc length of parametric curves. *IEEE Comput. Graph. Appl.*, 10(3):72–78, 1990. (Cited on page 9.)

[Gra85] C. Grant. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 79–84, New York, NY, USA, 1985. ACM Press. (Cited on pages 23 and 27.)

[GS87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987. (Cited on pages 39 and 50.)

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM. (Cited on page 44.)

[GWH01] M. Garland, A. Willmott, and P. Heckbert. Hierarchical face clustering on polygonal surfaces. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 49–58, New York, NY, USA, 2001. ACM. (Cited on page 39.)

[HA90] P. Haeberli and K. Akeley. The accumulation buffer: hardware support for high-quality rendering. *SIGGRAPH Comput. Graph.*, 24(4):309–318, 1990. (Cited on pages 18, 35, and 54.)

[Havoo] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. (Cited on pages 39 and 50.)

[HDKo1] S. Hiller, O. Deussen, and A. Keller. Tiled blue noise samples. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001*, pages 265–272. Aka GmbH, 2001. (Cited on pages 54, 92, and 110.)

[HDMSo3] V. Havran, C. Damez, K. Myszkowski, and H.-P. Seidel. An efficient spatio-temporal architecture for animation rendering. In P. Christensen, D. Cohen-Or, and S. Spencer, editors, *Rendering Techniques*, pages 106–117. Eurographics Association, 2003. (Cited on pages 30 and 50.)

[HSS97] W. Heidrich, P. Slusallek, and H.-P. Seidel. An image-based model for realistic lens systems in interactive computer graphics. In *Proceedings of the conference on Graphics interface '97*, pages 68–75, Toronto, Ont., Canada, Canada, 1997. Canadian Information Processing Society. (Cited on page 35.)

[IKoo] F. Dachille IX and A. Kaufman. High-degree temporal antialiasing. In *CA '00: Proceedings of the Computer Animation*, page 49, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on pages 17 and 28.)

[Jeno1] H. Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001. (Cited on pages 19 and 24.)

[JKo3] S. Joe and F. Kuo. Remark on algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 29(1):49–57, 2003. (Cited on pages 58 and 59.)

[JKo5] N. Jones and J. Keyser. Real-time geometric motion blur for a deforming polygonal mesh. In *CGI '05: Proceedings of the Computer Graphics International 2005*, pages 26–31, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 28.)

[Kaj86] J. Kajiya. The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, New York, NY, USA, 1986. ACM. (Cited on page 20.)

[KB83] J. Korein and N. Badler. Temporal anti-aliasing in computer generated animation. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 377–388, New York, NY, USA, 1983. ACM Press. (Cited on pages 18, 23, and 45.)

[Kelo1] A. Keller. Hierarchical Monte Carlo image synthesis. *Math. Comput. Simul.*, 55(1-3):79–92, 2001. (Cited on pages 31 and 34.)

[Kelo3] A. Keller. Strictly deterministic sampling methods in computer graphics. *SIGGRAPH 2003 Course Notes, Course #44: Monte Carlo Ray Tracing*, 2003. (Cited on pages 71 and 72.)

[Kel06] A. Keller. Myths of computer graphics. In H. Niederreiter, editor, *Monte Carlo and Quasi-Monte Carlo Methods 2004*, pages 217–243. Springer, 2006. (Cited on page 80.)

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. (Cited on page 44.)

[KH01] A. Keller and W. Heidrich. Interleaved sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 269–276, London, UK, 2001. Springer-Verlag. (Cited on page 18.)

[KK86] T. Kay and J. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM. (Cited on pages 39 and 45.)

[KK89] J. Kajiya and T. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 271–280, New York, NY, USA, 1989. ACM Press. (Cited on page 24.)

[KK02a] T. Kollig and A. Keller. Efficient bidirectional path tracing by randomized quasi-Monte Carlo integration. In H. Niederreiter, K. Fang, and F. Hickernell, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 290–305. Springer, 2002. (Cited on pages 67 and 75.)

[KK02b] T. Kollig and A. Keller. Efficient multidimensional sampling. *Computer Graphics Forum*, 21(3):557–563, September 2002. (Cited on pages 54, 62, 63, 73, 75, 77, and 85.)

[KK07] D. Kim and H.-S. Ko. Eulerian motion blur. In D. Ebert and S. Merillou, editors, *Eurographics Workshop on Natural Phenomena*, pages 39–46, Prague, Czech Republic, 2007. Eurographics Association. (Cited on pages 12 and 22.)

[KKS95] M.-J. Kim, M.-S. Kim, and S. Shin. A general construction scheme for unit quaternion curves with simple high order derivatives. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 369–376, New York, NY, USA, 1995. ACM Press. (Cited on page 10.)

[KMH95] C. Kolb, D. Mitchell, and P. Hanrahan. A realistic camera model for computer graphics. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 317–324, New York, NY, USA, 1995. ACM. (Cited on page 35.)

[Kno77] G. Knott. A numbering system for binary trees. *Commun. ACM*, 20(2):113–115, 1977. (Cited on page 41.)

[Knu97] D. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997. (Cited on page 41.)

[Knu05] D. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming)*. Addison-Wesley Professional, 2005. (Cited on page 41.)

[LAM01] J. Lext, U. Assarsson, and T. Möller. A benchmark for animated ray tracing. *IEEE Comput. Graph. Appl.*, 21(2):22–31, 2001. (Cited on page 24.)

[Lem08] C. Lemieux. *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer, 2008. (Cited on pages 53, 54, 58, and 75.)

[Lip79] W. Lipski. More on permutation generation methods. *Computing*, 23(4):357–365, 1979. (Cited on page 41.)

[LK81] B. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision (ijcai). In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 674–679, April 1981. (Cited on page 34.)

[Llo82] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982. (Cited on page 45.)

[LN86] R. Lidl and H. Niederreiter. *Introduction to finite fields and their applications*. Cambridge University Press, New York, NY, USA, 1986. (Cited on pages 57 and 58.)

[LP01] G. Larcher and F. Pillichshammer. Walsh series analysis of the L_2-discrepancy of symmetrisized point sets. *Monatsh. Math.*, 132:1–18, 2001. (Cited on pages 62, 63, and 92.)

[LV00] T. Lokovic and E. Veach. Deep shadow maps. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 385–392, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. (Cited on page 29.)

[LvBR93] J. Lucas, D. van Baronaigien, and F. Ruskey. On rotations and the generation of binary trees. *J. Algorithms*, 15(3):343–366, 1993. (Cited on page 41.)

[Max90] N. Max. Polygon-based post-process motion blur. *The Visual Computer*, 6(6):308–314, 1990. (Cited on page 30.)

[ML85] N. Max and D. Lerner. A two-and-a-half-d motion-blur algorithm. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 85–93, New York, NY, USA, 1985. ACM Press. (Cited on page 30.)

[Mooo0] Doug Moore. Fast Hilbert curve generation, sorting, and range queries, 2000. http://web.archive.org/web/20050204100342/http://www.caam.rice.edu/~dougm/twiddle/Hilbert/. (Cited on page 45.)

[Nei03] D. Neilson. Efficient algorithms in motion blurring and photon mapping. Master's thesis, University of Alberta, 2003. (Cited on page 20.)

[Neu07] I. Neulander. Pixmotor: a pixel motion integrator. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 23, New York, NY, USA, 2007. ACM. (Cited on page 30.)

[Nie87] H. Niederreiter. Point sets and sequences with small discrepancy. *Monatsh. Math.*, 104:273–337, 1987. (Cited on page 56.)

[Nie88] H. Niederreiter. Low-discrepancy and low-dispersion sequences. *Journal of Number Theory*, 30:51–70, 1988. (Cited on page 56.)

[Nie92] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. SIAM, Philadelphia, 1992. (Cited on pages 55, 56, 57, 62, 63, 67, and 68.)

[NX96a] H. Niederreiter and C. Xing. Low-discrepancy sequences and global function fields with many rational places. *Finite Fields and Their Applications*, 2:241–273, 1996. (Cited on pages 56 and 60.)

[NX96b] H. Niederreiter and C. Xing. Low-discrepancy sequences and global function fields with many rational places. *Finite Fields and Their Applications*, 2:241–273, 1996. (Cited on pages 56 and 60.)

[NY03] D. Neilson and Y.-H. Yang. Variance invariant adaptive temporal supersampling for motion blurring. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 67, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 19.)

[Ols07] J. Olsson. Ray-tracing time-continuous animations using 4d kd-trees. Master's thesis, Lund University, 2007. (Cited on page 51.)

[Owe95] A. Owen. Randomly permuted (t, m, s)-nets and (t, s)-sequences. In H. Niederreiter and P. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, volume 106 of *Lecture Notes in Statistics*, pages 299–315. Springer, 1995. (Cited on page 94.)

[Owe97] A. Owen. Monte Carlo variance of scrambled net quadrature. *SIAM J. on Numerical Analysis*, 34(5):1884–1910, 1997. (Cited on page 94.)

[PC83] M. Potmesil and I. Chakravarty. Modeling motion blur in computer-generated images. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 389–399, New York, NY, USA, 1983. ACM. (Cited on page 29.)

[PH04] M. Pharr and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. (Cited on pages 54, 62, and 92.)

[Pir02] G. Pirsic. A software implementation of Niederreiter-Xing sequences. In K.-T. Fang, F. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 434–445. Springer, 2002. (Cited on page 60.)

[Pro80] A. Proskurowski. On the generation of binary trees. *J. ACM*, 27(1):1–2, 1980. (Cited on page 41.)

[PS01] A. Pearce and K. Sung. US patent #6,211,882: Analytic motion blur coverage in the generation of computer graphics imagery, 2001. (Cited on page 24.)

[PT97] L. Piegl and W. Tiller. *The NURBS book (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1997. (Cited on pages 6 and 7.)

[PVTF02] W. Press, W. Vetterling, S. Teukolsky, and B. Flannery. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2002. (Cited on pages 58, 59, and 60.)

[Qua96] M. Quail. Space time ray tracing using ray classification, 1996. (Cited on page 51.)

[Ros07] G. Rosado. Motion blur as a post-processing effect. In H. Nguyen, editor, *GPU Gems 3*, chapter 27, pages 575–581. Addison-Wesley, July 2007. (Cited on page 30.)

[RP90] F. Ruskey and A. Proskurowski. Generating binary trees by transpositions. *J. Algorithms*, 11(1):68–84, 1990. (Cited on page 41.)

[RPH04] S. Ramsey, K. Potter, and C. Hansen. Ray bilinear patch intersections. *Journal of Graphics Tools*, 9(3):41–47, 2004. (Cited on page 24.)

[RV78] D. Rotem and Y. Varol. Generation of binary trees from ballot sequences. *J. ACM*, 25(3):396–404, 1978. (Cited on page 41.)

[SAG+05] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, and P. Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, Ltd., Natick, MA, USA, 2005. (Cited on pages 5, 17, and 30.)

[SD92] K. Shoemake and T. Duff. Matrix animation and polar decomposition. In *Proceedings of the conference on Graphics interface '92*, pages 258–264, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. (Cited on page 6.)

[Sed77] R. Sedgewick. Permutation generation methods. *ACM Comput. Surv.*, 9(2):137–164, 1977. (Cited on pages 41 and 81.)

[SF80] M. Solomon and R. Finkel. A note on enumerating binary trees. *J. ACM*, 27(1):3–5, 1980. (Cited on page 41.)

[Shi91] P. Shirley. Discrepancy as a quality measure for sample distributions. In *Eurographics '91*, pages 183–94. Elsevier Science Publishers, Amsterdam, North-Holland, 1991. (Cited on page 54.)

[Shi93] M. Shinya. Spatial anti-aliasing for animation sequences with spatio-temporal filtering. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 289–296, New York, NY, USA, 1993. ACM Press. (Cited on page 29.)

[Shi95] M. Shinya. Improvements on the pixel-tracing filter: Reflection/refraction, shadows, and jittering. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 92–102. Canadian Information Processing Society, Canadian Human-Computer Communications Society, 1995. ISBN 0-9695338-4-5. (Cited on page 29.)

[Sho85] K. Shoemake. Animating rotation with quaternion curves. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254, New York, NY, USA, 1985. ACM Press. (Cited on page 10.)

[Sho94] K. Shoemake. Polar matrix decomposition. In P. Heckbert, editor, *Graphics Gems IV*, pages 207–221. Academic Press, Boston, 1994. (Cited on page 6.)

[Sob67] I. Sobol'. On the distribution of points in a cube and the approximate evaluation of integrals. *Zh. vychisl. Mat. mat. Fiz.*, 7(4):784–802, 1967. (Cited on pages 56, 57, and 63.)

[Sob94] I. Sobol'. *A Primer for the Monte Carlo Method*. CRC Press, 1994. (Cited on page 17.)

[Spr92] R. Sprugnoli. The generation of binary trees as a numerical problem. *J. ACM*, 39(2):317–327, 1992. (Cited on page 41.)

[SPW02] K. Sung, A. Pearce, and C. Wang. Spatial-temporal antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 08(2):144–153, 2002. (Cited on pages 17, 23, 24, and 27.)

[ST90] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, New York, NY, USA, 1990. ACM. (Cited on page 35.)

[ST93] J. Shi and C. Tomasi. Good features to track. Technical report, Cornell University, Ithaca, NY, USA, 1993. (Cited on page 34.)

[Sta99] J. Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. (Cited on page 22.)

[Ste05] I. Stephenson. Shutter efficiency and temporal sampling. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches*, page 101, New York, NY, USA, 2005. ACM Press. (Cited on page 17.)

[TBI03] N. Tatarchuk, C. Brennan, and John R. Isidoro. Motion blur using geometry and shading distortion. In Wolfgang Engel, editor, *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Wordware, Plano, Texas, 2003. (Cited on page 28.)

[TK91] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991. (Cited on page 34.)

[TVC07] G. Takhtamyshev, B. Vandewoestyne, and R. Cools. Quasi-random integration in high dimensions. *Mathematics and Computers in Simulation*, 73(5):309–319, January 2007. (Cited on page 57.)

[Vea97] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1997. (Cited on page 21.)

[WABG06] B. Walter, A. Arbree, K. Bala, and D. Greenberg. Multidimensional lightcuts. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 1081–1088, New York, NY, USA, 2006. ACM Press. (Cited on pages 20 and 21.)

[Wäc08] C. Wächter. *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, Ulm University, 2008. (Cited on page 60.)

[Wal07] I. Wald. On fast construction of SAH based bounding volume hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007. (Cited on pages 39, 41, and 45.)

[War02] H. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Cited on page 60.)

[Wat70] G. Watkins. *A real time visible surface algorithm*. PhD thesis, The University of Utah, 1970. (Cited on pages 25 and 26.)

[WBS07] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1):6, 2007. (Cited on page 39.)

[WCE07] K. White, D. Cline, and P. Egbert. Poisson disk point sets by hierarchical dart throwing. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 129–132, September 2007. (Cited on page 54.)

[WFA+05] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. Greenberg. Lightcuts: a scalable approach to illumination. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1098–1107, New York, NY, USA, 2005. ACM. (Cited on page 20.)

[WGER05] D. Wexler, L. Gritz, E. Enderton, and J. Rice. GPU-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14, New York, NY, USA, 2005. ACM. (Cited on page 18.)

[WH06] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69, 2006. (Cited on page 39.)

[WK06] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006. (Cited on pages 49 and 50.)

[WMG+07] I. Wald, W. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. Parker, and P. Shirley. State of the art in ray tracing animated scenes. In *Eurographics 2007 State of the Art Reports*, 2007. (Cited on pages 48 and 49.)

[WZ96] M. Wloka and R. Zeleznik. Interactive real-time motion blur. *The Visual Computer*, 12(6):283–295, 1996. (Cited on page 28.)

[YCM07] S.-E. Yoon, S. Curtis, and D. Manocha. Ray tracing dynamic scenes using selective restructuring. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, page 55, New York, NY, USA, 2007. ACM. (Cited on pages 39 and 49.)

[Yel83] J. Yellot. Spectral consequences of photoreceptor sampling in the rhesus retina. *Science*, 221:382–385, 1983. (Cited on page 54.)

[YLVA00] S. Yakowitz, P. L'Ecuyer, and F. Vázquez-Abad. Global stochastic optimization with low-dispersion point sets. *Oper. Res.*, 48(6):939–950, 2000. (Cited on page 54.)

[YSQS07] L. Yuan, J. Sun, L. Quan, and H.-Y. Shum. Image deblurring with blurred/noisy image pairs. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 1, New York, NY, USA, 2007. ACM. (Cited on page 2.)

[Zag97] E. Scher Zagier. A human's eye view: motion blur and frameless rendering. *Crossroads*, 3(4):8–12, 1997. (Cited on pages 2 and 22.)

[Zer85] D. Zerling. Generating binary trees using rotations. *J. ACM*, 32(3):694–701, 1985. (Cited on page 41.)

[ZKTR06] Y. Zheng, H. Köstler, N. Thürey, and U. Rüde. Enhanced motion blur calculation with optical flow. In *Proceedings of the Vision, Modeling and Visualization 2006*, pages 253–260. Aka GmbH, 2006. (Cited on pages 29 and 34.)