

**SOFTWARE ALGORITHMS FOR
HARDWARE RAY TRACING**

by

Andrew E. Kensler

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2011

Copyright © Andrew E. Kensler 2011

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

The dissertation of _____ Andrew E. Kensler _____

has been approved by the following supervisory committee members:

_____ Steven G. Parker _____, Chair _____ April 25, 2010 _____
Date Approved

_____ Erik L. Brunvand _____, Member _____ _____
Date Approved

_____ Peter S. Shirley _____, Member _____ April 25, 2010 _____
Date Approved

_____ Ingo Wald _____, Member _____ April 25, 2010 _____
Date Approved

_____ Claudio T. Silva _____, Member _____ _____
Date Approved

and by _____ Martin Berzins _____, Chair of

the Department of _____ School of Computing _____

and by David Chapman, Dean of The Graduate School.

ABSTRACT

This dissertation explores three key facets of software algorithms for custom hardware ray tracing: primitive intersection, shading, and acceleration structure construction. For the first, primitive intersection, we show how nearly all of the existing direct three-dimensional (3D) ray-triangle intersection tests are mathematically equivalent. Based on this, a genetic algorithm can automatically tune a ray-triangle intersection test for maximum speed on a particular architecture. We also analyze the components of the intersection test to determine how much floating point precision is required and design a numerically robust intersection algorithm. Next, for shading, we deconstruct Perlin noise into its basic parts and show how these can be modified to produce a gradient noise algorithm that improves the visual appearance. This improved algorithm serves as the basis for a hardware noise unit. Lastly, we show how an existing bounding volume hierarchy can be postprocessed using tree rotations to further reduce the expected cost to traverse a ray through it. This postprocessing also serves as the basis for an efficient update algorithm for animated geometry. Together, these contributions should improve the efficiency of both software- and hardware-based ray tracers.

To Marian, Daniel and Veronica.

CONTENTS

ABSTRACT	iii
LIST OF TABLES	vii
LIST OF FIGURES	viii
ACKNOWLEDGMENTS	x
CHAPTERS	
1. INTRODUCTION	1
1.1 Thesis Statement	3
1.2 Motivation	3
2. BACKGROUND AND RELATED WORK	5
2.1 A Simple Ray Tracer	5
2.2 Noninteractive Ray Tracing	21
2.3 Interactive Ray Tracing	23
2.4 Custom Hardware for Interactive Ray Tracing	26
3. OPTIMIZING RAY-TRIANGLE INTERSECTION VIA AUTOMATED SEARCH	28
3.1 Introduction	29
3.2 Signed Volumes	30
3.3 Using Signed Volumes For Intersection	32
3.4 Minimizing Total Operation Counts	34
3.5 A Genetic Algorithm for Improved Performance	37
3.6 General Packet Code	41
3.7 Numerical Robustness	46
3.8 Significand and Exponent Widths	49
3.9 Robustly Computing Signed Volumes	53
3.10 Designing a Robust Algorithm	55
3.11 Conclusion	60
4. BETTER GRADIENT NOISE	61
4.1 Introduction	62
4.2 Noise in Graphics	63
4.3 Noise in Hardware	68
4.4 Gradient Vectors	70

4.5	Hash Function	71
4.6	Filter Kernel	73
4.7	Projection to 2D	76
4.8	Software Algorithm	79
4.9	Hardware Gradient Noise	80
4.10	Lookup Tables	84
4.11	Pipelining	85
4.12	Physical Implementation	86
4.13	Conclusion	87
5.	IMPROVING BOUNDING VOLUME HIERARCHIES THROUGH TREE ROTATIONS	90
5.1	Introduction	91
5.2	Tree Rotations	95
5.3	Hill Climbing	97
5.4	Simulated Annealing	97
5.5	Performance on Static Geometry	101
5.6	Update Algorithm	104
5.7	Performance on Animations	106
5.8	Modifying the Scene Graph	111
5.9	Conclusion	114
6.	CONCLUSION	115
6.1	Subsequent Work	117
6.2	Future Work	118
	REFERENCES	121

LIST OF TABLES

3.1	Performance comparison of ray-triangle intersection algorithms. . . .	41
3.2	Relative error for determinant by significand and exponent widths..	50
3.3	Relative error for distance by significand and exponent widths. . . .	51
3.4	Relative error for barycentrics by significand and exponent widths. .	52
3.5	Comparison on Stanford Bunny with various significand widths. . .	58
3.6	Comparison on Sponza Atrium with various significand widths. . . .	59
4.1	Cell area and performance of synthesized noise module.	85
5.1	Results of tree rotations on static scenes.	103
5.2	Results for tree rotations on animations.	108

LIST OF FIGURES

2.1	Geometry of a ray.	9
2.2	Three common acceleration structures	11
2.3	Pinhole camera model and assembly of a ray packet.	19
3.1	Determinants and signed areas or volumes.	30
3.2	Homegenous coordinates and triangle area.	31
3.3	Geometry for a ray edge test.	33
3.4	Volumes for all ray edge tests.	33
3.5	Distribution of values by floating point exponent.	48
3.6	Pseudocode for robust ray-triangle intersection algorithm.	56
4.1	Rendered scene exhibiting noise-based procedural textures.	64
4.2	Value noise and Perlin's gradient noise.	66
4.3	Simple scene comparing three noise algorithms.	69
4.4	Projections of 3D gradient vectors onto planar cross section.	72
4.5	Traditional and <i>xor</i> hash functions.	74
4.6	Separable and radial filter kernels.	76
4.7	Gradient noise with extended reconstruction kernel.	76
4.8	Low frequencies in planar slice.	77
4.9	Projecting the lattice points.	78
4.10	Low frequencies removed by projection.	78
4.11	Pseudocode for the complete noise algorithm.	79
4.12	Noise evaluated on half-resolution grid.	81
4.13	High-level diagram of our hardware noise implementation.	82
4.14	Rendered scene with fixed point version of our noise.	83
4.15	Placed and routed circuit implementing our improved noise.	87
5.1	Left- and right-rotations on ordered binary trees	95
5.2	Nodes to exchange for each possible rotation	96
5.3	Progressive lowering of SAH cost for conference room	98

5.4 Simulated annealing applied to the Soda Hall model	100
5.5 Render times and SAH costs on the DragBun animation.	109
5.6 Break-down of time spent on the on the DragBun animation.	110
5.7 Comparison on BART scene with long and short animation loops. . .	112

ACKNOWLEDGMENTS

The models used in this thesis were provided by a variety of sources. The Stanford Bunny, Happy Buddha, and Dragon models were provided by the Stanford Computer Graphics Laboratory's 3D Scanning Repository. The Sponza Atrium scene was created by Marko Dabrovic of RNA Studio. The sections of the Power Plant Model were provided by the Walkthru Group at UNC who recieved it from an anonymous donor. The LBL Building 90 Third Floor Conference Room scene was originally modelled by Anat Grynberg and Greg Ward for the Radiance rendering system. The U.C. Berkeley Soda Hall WALKTHRU Model is courtesy of Celeste Fowler, Tom Funkhouser, Seth Teller, and Carlo Sequin.

The Fairy Forest animation was created by Ingo Wald using DAZ Studio with elements provided by DAZ Productions. Three of the animations: the Exploding Dragon and Bunny, the Cloth Simulation, and the N-Body Simulation are from the UNC Dynamic Scene Benchmarks. The BART animation is a subset of the Museum test scene from the Benchmark for Animated Ray Tracing [57].

Parts of this work were supported by NSF grant 05-41009 and undertaken for the University of Utah Hardware Ray Tracing project.

I wish to thank my colleagues, coauthors and committee members, without whom this thesis would not exist. Of these, I particularly wish to thank Steve Parker and Erik Brunvand: Steve for being my official advisor, and Erik for being my de facto coadvisor. Their funding and advice made this work possible.

Above all, I wish to thank my wife, Marian, and my children, Daniel and Veronica. Their support has kept me going through the ups and downs over the years.

CHAPTER 1

INTRODUCTION

Automatic computers have been ray tracing geometric optics almost from the beginning of their history [35]. These early computers performed ray tracing to assist lens makers with designing new lenses. Before that, lens makers had to resort to tedious hand computations to trace light paths through their lens designs. One such early system took 115 seconds to trace a path through eleven surfaces [36].

Ray queries, the core concept of ray tracing is relatively simple in principle: for a given position and direction (a ray), look through the database and determine the closest point along the ray that it intersects with the surface of an object. Broadly speaking, rays in a ray tracer move through four phases: ray generation, acceleration structure traversal, primitive intersection, and shading. Ray generation creates the rays for each pixel in the image and determines their origin and direction. Acceleration structure traversal produces candidate sets of primitives that may have intersections with each ray. Primitive intersection tests the rays against these candidates to determine the nearest intersection, if any. Finally, shading determines how the ray's state affects the appearance of the image, and may in fact, generate new rays.

Rasterization is the major alternative to ray tracing. A rasterizer takes each primitive and projects it onto the image. For each pixel that the primitive covers, it checks to see if the primitive would be visible at that point (i.e., closer than any opaque surfaces that might hide it) and if so it updates the pixel. The classic rasterization method for polygons begins by clipping each polygon to the viewing frustum, performs scanline conversion to determine spans of covered

pixels and then uses the Z-buffer algorithm to test visibility at each pixel.

The Reyes algorithm [18] is a more advanced form of rasterization. It estimates the projected area on the image for each primitive. If the primitive is completely outside of the image it discards it. If the area is above a certain threshold then it recursively splits it and projects the halves onto the image. When the pieces are small enough, it dices them into grids of subpixel sized micropolygons and sends them to an A-buffer [14] to test their visibility and add their contribution to the image.

In their purest forms, ray tracers and rasterizers are duals of each other: a ray tracer's outer loop processes pixels and its inner loop searches through primitives. In a rasterizer these loops are exchanged: its outer loop processes primitives and its inner loop searches through pixels.

Rasterization's inner loop is highly efficient at updating pixels on the screen which has made it the algorithm of choice for commercial graphics processing units (GPUs). Its major drawback is that it can only directly handle local light transport effects. There are ways of achieving effects such as shadows, planar reflections, and so forth, but they can take significant effort and combining these techniques is nontrivial. In contrast, a ray tracer is well tuned for the searches through the primitives. This makes it a useful component for rendering global effects such as specular reflections and refractions, soft shadows, diffuse interreflections, caustics, and depth of field. Furthermore, ray tracing allows all of these effects to be combined cleanly. Even systems based on rasterization for the primary visibility method can benefit from the addition of ray tracing to provide these global effects.

While such hybrid rendering systems are moderately common for software-based batch renderers, they are quite uncommon in hardware accelerated interactive systems; today's GPUs are principally designed to rasterization and require a number of tricks to render global effects. Extending this model into a hybrid system by the addition of new circuitry designed to accelerate ray queries should offer the best of both worlds.

Custom hardware affords us the opportunity to rethink the ray tracing software that we write. Are there ways to improve the speed of ray-triangle intersections or acceleration structure traversal? Can we reduce the memory traffic between the host system and the rendering device? We believe that the answer is yes, and we will explore three areas for improvements: basic ray-triangle intersection, noise-based procedural texturing, and the optimization and maintenance of bounding volume hierarchy (BVH) trees. Each of these has mathematical properties that offer an avenue of attack.

1.1 Thesis Statement

Exploring the mathematical structure of frequently performed operations in ray tracing kernels can lead to improved algorithms for ray-triangle intersection and scene BVH construction by employing stochastic optimization algorithms. Analyzing Perlin noise in the frequency domain can lead to higher visual quality for procedural textures by controlling aliasing and structured artifacts. These algorithms improve the efficiency of software- and hardware-based ray tracers.

1.2 Motivation

The work on triangle intersection algorithms grew from dissatisfaction with the existing direct 3D ray-triangle tests. With so many to choose from, how could we be sure that we hadn't missed a better one? After discovering the fundamental mathematical equivalence of a good number of these test, the next step was to try to explore as much as possible of the unified design space for these algorithms.

The Utah Hardware Ray Tracing group's TRaX project has been the principal motivation for the work on noise algorithms and tree rotations. The research into noise started from the observation that ray tracing hardware would likely be limited by main memory bandwidth. One way to reduce that was to reduce the amount of image-based textures by relying more on procedural texturing, which led naturally to the desire for a hardware noise unit. The initial study in this area led not to more efficient hardware, but rather to a slightly slower

software algorithm that improved on the visual quality. The understanding gained from this intermediate step finally led to the design of hardware unit.

The work on tree rotations began with the question of how close the surface area heuristic (SAH) based BVH construction algorithm was to optimal and whether there was any way to build a BVH and then feed back the information derived from the construction of that tree to produce a better one. The subsequent work on tree rotations for animated scenes was done with the goal of producing an update algorithm that could run efficiently on the TRaX architecture.

CHAPTER 2

BACKGROUND AND RELATED WORK

The first part of this chapter provides an overview of interactive ray tracing. Step-by-step it presents the source code for a small, but working interactive ray tracer and explains how it works in terms of the basic algorithms used in ray tracing.

Following this is a broad overview of related work on the general topics of ray tracing and interactive ray tracing. The following three chapters will all also discuss more specific areas of related work.

2.1 A Simple Ray Tracer

In order to illustrate in detail how interactive ray tracing works, we will present and describe the code for Mirth, a miniature interactive ray tracer. Though compact at approximately 350 lines of readable, portable C++, it can render the 69,451 triangle Stanford Bunny model at up to 11 frames per second on a 2.8 GHz Core 2 Duo processor with the bunny nearly filling the 512×512 frame.

We will describe the code piece by piece. The first part simply includes the headers and defines two data types:

```
1 #include <float.h>
2 #include <math.h>
3 #include <fstream>
4 #include <algorithm>
5 #ifdef __APPLE__
6 #include <GLUT/glut.h>
7 #else
8 #include <GL/glut.h>
9 #endif
10 using namespace std;
```



```

11
12 struct vec {
13     float data[3];
14     inline vec() {}
15     inline vec(float const x, float const y, float const z) {
16         data[0] = x; data[1] = y; data[2] = z;
17     }
18     inline float &operator[](int const index) {
19         return data[index];
20     }
21     inline float const &operator[](int const index) const {
22         return data[index];
23     }
24     inline vec operator+(vec const &right) const {
25         return vec(data[0] + right[0],
26                   data[1] + right[1],
27                   data[2] + right[2]);
28     }
29     inline vec operator-(vec const &right) const {
30         return vec(data[0] - right[0],
31                   data[1] - right[1],
32                   data[2] - right[2]);
33     }
34     inline vec operator*(float const right) const {
35         return vec(data[0]*right, data[1]*right, data[2]*right);
36     }
37     inline vec minimum(vec const &right) const {
38         return vec(min(data[0], right[0]),
39                   min(data[1], right[1]),
40                   min(data[2], right[2]));
41     }
42     inline vec maximum(vec const &right) const {
43         return vec(max(data[0], right[0]),
44                   max(data[1], right[1]),
45                   max(data[2], right[2]));
46     }
47     inline float dot(vec const &right) const {
48         return data[0]*right[0] + data[1]*right[1] + data[2]*right[2];
49     }
50     inline vec cross(vec const &right) const {
51         return vec(data[1]*right[2] - data[2]*right[1],
52                   data[2]*right[0] - data[0]*right[2],
53                   data[0]*right[1] - data[1]*right[0]);
54     }
55     inline vec normalize() const {
56         return *this*(1.0f / sqrt(dot(*this)));
57     }
58 };
59
60 typedef vec rgb;

```

The first data type, `vec`, is a three-dimensional vector class. Operator overloading is used for indexing components, vector addition, subtraction, and scaling. It also defines methods for finding the component-wise minimum and maximums, computing the dot product, computing the cross product, and normalizing the vector to unit length.

The second type defined, `rgb`, is used to represent colors. It is just an alias for the vector type.

Next, we have the code for handling the model to render:

```

62 struct tri {
63     vec edge_0, edge_1, corner;
64     rgb color;
65     inline tri() {}
66 } *tris;
67
68 int read_model(char const *filename) {
69     ifstream in(filename);
70     int num_tris;
71     in >> num_tris;
72     tris = new tri[num_tris];
73     for (int t = 0; t < num_tris; ++t) {
74         vec verts[3];
75         for (int v = 0; v < 3; ++v)
76             for (int c = 0; c < 3; ++c)
77                 in >> verts[v][c];
78         tris[t].edge_0 = verts[1] - verts[0];
79         tris[t].edge_1 = verts[2] - verts[0];
80         tris[t].corner = verts[0];
81         for (int c = 0; c < 3; ++c)
82             in >> tris[t].color[c];
83     }
84     return num_tris;
85 }

```

Mirth uses triangles as its sole rendering primitive. Triangles, represented by the `tri` class, are stored with two edges pointing away from a common corner. They also store the color that the triangle should appear in. A model is simply stored as a flat array of `tri` objects accessible through the `tris` pointer.

The `read_model()` function reads in the model to render from a simple text file. The first number given in the file is an integer count of the number of

triangles to follow. Each following triangle is given by twelve floating point numbers: three for each of the three corners, followed by three more between zero and one giving the red, green and blue values for the triangle's color. The `read_model()` function allocates enough space to hold all of these, and points `tris` to the list of triangles that it read.

Given that this is a ray tracer, we also need to define what a ray is:

```

87 struct ray {
88     vec orig, dir, inv;
89     int signs[3];
90     float t;
91     int tri_index;
92     rgb color;
93 };

```

Rays are defined parametrically with a point and a direction: $\mathbf{p} = \mathbf{o} + t\mathbf{d}$. The point, \mathbf{o} , defines where the ray begins, and is called the origin. The direction, \mathbf{d} , determines which way the ray points towards. Points along the ray are numbered by the ray parameter, t . The core function of a ray tracer is finding the smallest positive t that marks a point on the surface of one of the objects in the scene. The `orig` and `dir` fields store the ray origin and direction, respectively, and the ray parameter of any intersection found is stored in `t`.

Figure 2.1 illustrates this. After testing this ray for intersections with the scene geometry, point (a) at $t = 2$ would be returned as the closest intersection. While point (b) with $t = 3$ would also be considered a "hit," it would not be returned because it was farther along the ray than (a). At point (c), the ray comes close to the sphere but misses it. Intersection (d) at $t = -1$ would also be reported as a miss because the point is behind the ray origin.

The `inv` vector and `signs` array store information about the ray's direction vector. The `inv` vector contains the reciprocals of each component of the ray direction. The `signs` array corresponds to the direction and stores a one for a negative component and a zero for a nonnegative component. These are precomputed and cached to speed up the intersection tests.

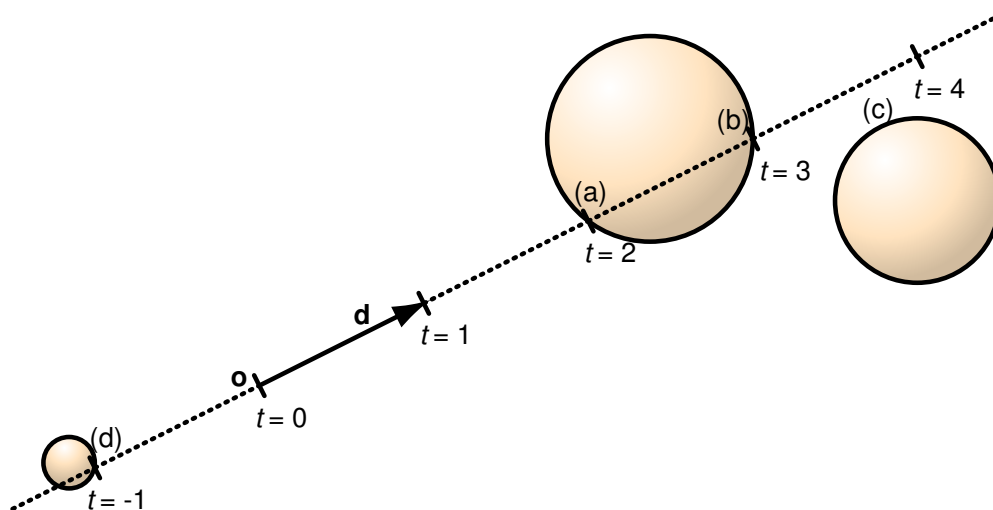


Figure 2.1. Geometry of a ray.

If the ray should hit a triangle, then in addition to updating t , the `tri_index` is updated with the index of the intersected triangle in the triangle list. The shading process uses both of those to compute the color and intensity of the light returned along the ray and stores the result in the `color` field.

Before that, however, we still have to find the closest intersection with a ray:

```

95 static float const epsilon = 0.001f;
96
97 inline void ray_tri_int(ray *rays, int const num_rays,
98                       int const first, int const last) {
99     for (int ti = first; ti < last; ++ti) {
100         tri const &t(tris[ti]);
101         vec norm(t.edge_0.cross(t.edge_1));
102         for (int ri = 0; ri < num_rays; ++ri) {
103             ray &r(rays[ri]);
104             float det = r.dir.dot(norm);
105             vec offset(r.orig - t.corner);
106             vec cross(r.dir.cross(offset));
107             float u = t.edge_1.dot(cross);
108             float v = t.edge_0.dot(cross);
109             if (u*det < 0.0f || v*det > 0.0f || fabs(u - v) > fabs(det))
110                 continue;
111             float recip = 1.0f / det;
112             float t_val = offset.dot(norm)*-recip;
113             if (t_val > epsilon && t_val < r.t) {

```

```

114         r.t = t_val;
115         r.tri_index = ti;
116     }
117 }
118 }
119 }

```

To find the intersection of a ray with a triangle, $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$, we can define points in space as a weighted sum of the triangle's vertices: $\mathbf{p} = \alpha\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2$. The weights, α , β and γ are known as barycentric coordinates and for points on the triangle they are nonnegative and sum to one. The later constraint lets us eliminate one of the barycentric coordinates: $\alpha = 1 - \beta - \gamma$. Taking the intersection as the point that lies on both the triangle and on the ray gives:

$$(1 - \beta - \gamma)\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2 = \mathbf{o} + t\mathbf{d}.$$

A bit of rearrangement and expansion produces a 3×3 linear system to solve:

$$\begin{bmatrix} \mathbf{p}_{1x} - \mathbf{p}_{0x} & \mathbf{p}_{2x} - \mathbf{p}_{0x} & -\mathbf{d}_x \\ \mathbf{p}_{1y} - \mathbf{p}_{0y} & \mathbf{p}_{2y} - \mathbf{p}_{0y} & -\mathbf{d}_y \\ \mathbf{p}_{1z} - \mathbf{p}_{0z} & \mathbf{p}_{2z} - \mathbf{p}_{0z} & -\mathbf{d}_z \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{o}_x - \mathbf{p}_{0x} \\ \mathbf{o}_y - \mathbf{p}_{0y} \\ \mathbf{o}_z - \mathbf{p}_{0z} \end{bmatrix}$$

We can solve this numerically using Cramer's rule. For conciseness, we compute the matrix determinants using the scalar triple product. From there we simply check that the barycentric coordinates define a point inside the triangle (i.e., $\beta \geq 0$, $\gamma \geq 0$, and $\beta + \gamma \leq 1$), and that the ray parameter is positive. This is the essence of the Möller-Trumbore [63] algorithm. Solving this linear system efficiently and robustly is the topic of Chapter 3.

The code for `ray_tri_int()` takes a list of rays and a range of triangles, and tests each ray against each triangle. When it finds a valid intersection closer than the closest one already found, it updates the ray's `t` and `tri_index` fields. Note that we make sure that the ray parameter is greater than a small constant, ϵ . This prevents rays leaving a surface from immediately intersecting that surface again. Without this, the ray tracer can produce an artifact known as surface acne.

Using `ray_tri_int()` to test every ray against every triangle is sufficient, but slow. To make it quick, we need to use an acceleration structure. These are auxiliary data structures that are precomputed once for the model and then used to quickly derive a small set of candidate triangles. Figure 2.2 shows the three acceleration structures most commonly used today.

Bounding volume hierarchies, or BVHs [84], use a primitive that is cheap to test for intersection, such as a box, as a proxy for a piece of the actual model [17]. The proxy, or bounding volume, completely encloses its piece of the model; if the ray misses the bounding volume, then it obviously misses the actual model inside. If it hits, then we may need to test the ray against the actual geometry. BVHs nest these bounding volumes within each other like a set of matryoshka dolls. Together these form a tree of bounding volumes, with the model's primitives at the leaves. The BVH is an “object subdivision” acceleration structure – every object belongs to a single leaf, but a point in space can belong to multiple nodes.

Instead of subdividing the objects, however, we can form a tree by subdividing the space itself. This is how the kd-tree [9], a “spatial subdivision” structure works. The space itself is subdivided into smaller and smaller pieces by axis-aligned planes. Each point in space belongs to a single node, but now the scene primitives can belong to multiple nodes. To trace a ray, we find the

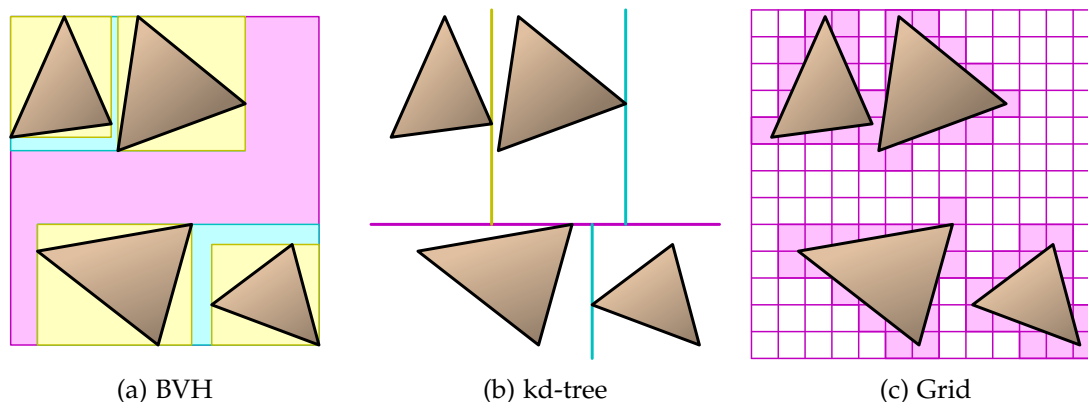


Figure 2.2. Three common acceleration structures

node that contains the ray's origin and then walk the tree, moving up and down as the ray crosses the planar partitions between regions. At each leaf we test the ray for intersections with any associated primitives that overlap the region.

The grid [30] is another spatial subdivision acceleration structure, but it is nonhierarchical. In a uniform grid, space is divided up into fixed, regular sized cells that store a list of primitives that touch it. To trace a ray through the grid, we simply step along the ray through the grid, cell by cell, beginning with the cell closest to the origin. At each step, we look up any primitives attached to the cell and test them against the ray for intersections.

Each of these makes the typical cost to intersect a ray with the scene sublinear with respect to the number of primitives. This is one of the great advantages that ray tracing has when scene database gets very large.

Of these three acceleration structures, Mirth uses the BVH with axis-aligned boxes as the bounding volume. First, we will define a short helper function that operates on the triangle list:

```

121 inline int partition(int first, int last,
122                   int const axis, float const position) {
123     for (--first; ;) {
124       for (++first;
125           (tris[first].corner[axis] +
126            (tris[first].edge_0[axis] +
127             tris[first].edge_1[axis])) / 3.0f <= position &&
128           first < last);
129         ++first);
130       for (--last;
131           (tris[last].corner[axis] +
132            (tris[last].edge_0[axis] +
133             tris[last].edge_1[axis])) / 3.0f >= position &&
134           first < last);
135         --last);
136       if (first >= last)
137         return first;
138       tri const t(tris[first]);
139       tris[first] = tris[last];
140       tris[last] = t;
141     }
142 }

```

This interesting function is almost identical to the partition function used in the classic quicksort algorithm. Rather than a scalar pivot, however, it uses an axis-aligned plane, specified by axis and position. After a call to this `partition()` function, all of the triangles in the list between `first`, inclusive, and `last`, exclusive, will be partitioned in-place into two groups based on which side of an axis-aligned plane their centroids fall on. Triangles with centroids having a smaller value on the axis will be moved towards the beginning of the range, while those with a greater value will be moved towards the end. The function returns the index of the triangle that begins this second group.

The `partition()` function is used by the code to construct the bounding volume hierarchy:

```

144 struct node {
145     vec box[2];
146     unsigned int index;
147     unsigned int last : 29;
148     unsigned int axis : 2;
149     bool leaf : 1;
150 } *bvh;
151
152 static int const max_tris_per_leaf = 4;
153
154 int build_bvh(int const index, int const size,
155             int const first, int const last) {
156     node &n(bvh[index]);
157     n.box[0] = tris[first].corner;
158     n.box[1] = tris[first].corner;
159     for (int t = first; t < last; ++t) {
160         n.box[0] = n.box[0].minimum(tris[t].corner)
161                 .minimum(tris[t].corner + tris[t].edge_0)
162                 .minimum(tris[t].corner + tris[t].edge_1);
163         n.box[1] = n.box[1].maximum(tris[t].corner)
164                 .maximum(tris[t].corner + tris[t].edge_0)
165                 .maximum(tris[t].corner + tris[t].edge_1);
166     }
167     if (last - first < max_tris_per_leaf) {
168         n.index = first;
169         n.last = last;
170         n.leaf = true;
171         return size;
172     }
173     vec diag(n.box[1] - n.box[0]);
174     n.axis = ((diag[0] > diag[1] &&
```



```

175         diag[0] > diag[2]) ? 0 :
176         (diag[1] > diag[2]) ? 1 : 2);
177     float position = (n.box[0][n.axis] + n.box[1][n.axis])*0.5f;
178     int split = partition(first, last, n.axis, position);
179     if (split == first || split == last)
180         split = (first + last) / 2;
181     n.index = size;
182     n.leaf = false;
183     int new_size = build_bvh(size, size + 2, first, split);
184     new_size = build_bvh(size + 1, new_size, split, last);
185     return new_size;
186 }

```

The BVH construction is recursive. To build a node for a range of triangles, we start by finding the smallest axis-aligned box that contains all of the triangles. The box is represented by a pair of opposite corners, stored in the `box`. If there are fewer than `max_tris_per_leaf` in this node, we flag it as a leaf, and store the indices to the triangles in `index` and `last`.

Otherwise, we find and store the axis that the box is longest in, locate the box's midway point along that axis, and call `partition()` to subdivide the triangles into two sets – one to each side of that midway point. We mark the node as an interior node, set `index` to the index of the next free BVH node and recursively call `build_bvh()` to build the child nodes for each group. If the partition function failed to subdivide the group for some reason, it arbitrarily splits the triangle list in half and hopes to do better on the children.

This particularly strategy for subdividing the primitives at a node is sometimes called the longest-axis spatial-median split algorithm. It is quite simple and moderately effective. Chapter 5 will discuss a more sophisticated BVH construction strategy known as the surface area heuristic.

The next function, `ray_box_int()` tests a set of rays against a node to see if any of them intersect its box:

```

188 inline bool ray_box_int(ray *&rays, int &num_rays, int const index) {
189     node const &n(bvh[index]);
190     for (int ri = 0; ri < num_rays; ++ri) {
191         ray const &r(rays[ri]);

```

```

192     float max_min = epsilon;
193     float min_max = r.t;
194     float x_min = (n.box[r.signs[0]][0] - r.orig[0])*r.inv[0];
195     float x_max = (n.box[1 - r.signs[0]][0] - r.orig[0])*r.inv[0];
196     float y_min = (n.box[r.signs[1]][1] - r.orig[1])*r.inv[1];
197     float y_max = (n.box[1 - r.signs[1]][1] - r.orig[1])*r.inv[1];
198     float z_min = (n.box[r.signs[2]][2] - r.orig[2])*r.inv[2];
199     float z_max = (n.box[1 - r.signs[2]][2] - r.orig[2])*r.inv[2];
200     if (min_max < x_min || x_max < max_min)
201         continue;
202     min_max = min(min_max, x_max);
203     max_min = max(max_min, x_min);
204     if (min_max < y_min || y_max < max_min)
205         continue;
206     min_max = min(min_max, y_max);
207     max_min = max(max_min, y_min);
208     if (min_max >= z_min && z_max >= max_min) {
209         rays += ri;
210         num_rays -= ri;
211         return true;
212     }
213 }
214 return false;
215 }

```

Each of face of the box defines a plane, and each of the planes defines a half-space. The box is simply the intersection of all six half spaces. If we think of the planes in terms of three pairs, one for each axis, the intersection of a pair of half-spaces forms a “slab” in space. A ray that intersects the slab will enter by one face and leave from the other, defining an interval of ray parameter values where it is inside the slab. The intersection of the three intervals gives the interval in which the ray passes inside the box. If the intersection is null then the ray misses the box. This implementation also test to see that intersection overlaps the interval between epsilon and the preexisting closest hit. If the ray’s intersection with the box is behind the origin, or if the closest point is farther than a hit we have already found, there is no need to pursue that node of the BVH further.

The implementation used here is based on the Williams et al. [104] algorithm. The implementation here puts it in a loop over a set of rays and borrows an idea from Wald et al. [97] – when it finds a ray that strikes the box, it reduces the

range of rays that it was passed by the number of rays that failed to hit it. This way, it will not waste any more time testing those rays again as it descends the BVH during traversal:

```

217 inline void intersect(ray *rays, int num_rays, int const index) {
218     if (!ray_box_int(rays, num_rays, index))
219         return;
220     node const &n(bvh[index]);
221     if (n.leaf)
222         ray_tri_int(rays, num_rays, n.index, n.last);
223     else {
224         intersect(rays, num_rays, n.index + rays[0].signs[n.axis]);
225         intersect(rays, num_rays, n.index + 1 - rays[0].signs[n.axis]);
226     }
227 }

```

With the machinery for ray/triangle and ray/box intersection set up before, traversing a ray through the BVH is almost trivial. The `intersect()` function simply recurses through the BVH. If all of the rays miss the box around the node then it stops there and returns. Otherwise, depending on whether the node is a leaf or not, it either tests the rays against the triangles, or recursively calls itself on the two children. Note that it checks the sign of one of the components of the ray direction to determine which to visit first. If it can find an intersection in the nearer child, it may not have to descend much into the further child.

Once `intersect()` returns, all of the rays will have had their `t` and `tri_index` fields set to the closest valid intersection. The next step is to use this information to compute a color for each ray:

```

229 inline void set_ray(ray &r, float const max_t,
230                   vec const &orig, vec const &dir) {
231     r.t = max_t;
232     r.orig = orig;
233     r.dir = dir;
234     r.inv = vec(1.0f / dir[0], 1.0f / dir[1], 1.0f / dir[2]);
235     for (int c = 0; c < 3; ++c)
236         r.signs[c] = dir[c] < 0.0f;
237 }
238
239 vec light(0.9f, 0.9f, 3.0f);

```

```

240
241 static int const packet_width = 2;
242
243 inline void shade(ray *rays, int const num_rays) {
244     int map[packet_width*packet_width], num = 0;
245     ray s[packet_width*packet_width];
246     for (int ri = 0; ri < num_rays; ++ri) {
247         ray &r(rays[ri]);
248         r.color = rgb(0.0f, 0.0f, 0.0f);
249         if (r.t >= FLT_MAX)
250             continue;
251         tri const &t(tris[r.tri_index]);
252         r.color = t.color*0.1f;
253         vec norm(t.edge_0.cross(t.edge_1));
254         vec hit_pos(r.orig + r.dir*r.t);
255         vec light_dir(light - hit_pos);
256         float illum = norm.dot(light_dir);
257         if (illum < 0.0f)
258             continue;
259         illum /= sqrt(light_dir.dot(light_dir))*sqrt(norm.dot(norm));
260         map[num] = ri;
261         set_ray(s[num], 1.0f, hit_pos, light_dir);
262         s[num++].color = t.color*(illum*0.9f);
263     }
264     intersect(s, num, 0);
265     for (int ri = 0; ri < num; ++ri)
266         if (s[ri].t >= 1.0f - epsilon)
267             rays[map[ri]].color = rays[map[ri]].color + s[ri].color;
268 }

```

The shade() function implements the Lambertian model,

$$C(K_d \max(0, \frac{\mathbf{L} \cdot \mathbf{N}}{|\mathbf{L}||\mathbf{N}|}) + K_a),$$

where C is the base color of the surface defined by the hit triangle's color field, \mathbf{N} is the triangle's surface normal, and \mathbf{L} is the vector from the hit point towards the light source at position `light`. The K_d coefficient, fixed at 0.9 here, controls the contribution from the diffuse term, and the K_a coefficient, fixed at 0.1, controls the ambient.

While this program uses a constant surface color, C , for each triangle, a more advanced renderer would use the barycentric coordinates of the triangle at the intersection to vary the color across the triangle. Procedural texturing computes

the color algorithmically and often uses a noise function as a building block. Noise functions will be discussed in detail in Chapter 4.

The implementation here is relatively straight forward, looping over each ray and computing the terms. The only unusual part is that for each ray whose hit point faces towards the light source, we use `set_ray()` to build a new ray from the hit point towards the light source and record the diffuse term in new ray. Then we call `intersect()` on this new bundle of rays and for each of these that makes it to the light source without an intersection, we add the stored diffuse term to the original ray's color. This way, we trace additional rays to perform the visibility test for hard shadows.

The `packet_width` constant is only used by `shade()` to ensure that it statically allocates enough space to be able trace a shadow ray for each ray in the group that it receives from the `render()` function:

```

270 vec eye(0.0f, 0.0f, 0.0f), look_at(0.02f, 0.0f, 0.1f);
271 float yaw(0.0f), pitch(0.0f), track(0.3f), fov(35.0f);
272 rgb *image;
273 int mouse_x, mouse_y, buttons;
274
275 static int const image_size = 512;
276 static int const packets_per_row = image_size / packet_width;
277
278 void render() {
279     light = vec(light[0]*cos(0.1f) + light[1]*sin(0.1f),
280               light[0]*-sin(0.1f) + light[1]*cos(0.1f),
281               light[2]);
282     float scale = 2.0f*tan(0.5f*fov*3.14159265349f/180.0f) / image_size;
283     vec w((look_at - eye).normalize());
284     vec u(w.cross(vec(0.0f, 0.0f, 1.0f)).normalize()*scale);
285     vec v(u.cross(w).normalize()*scale);
286     vec c(w - (u + v)*image_size*0.5f);
287     #pragma omp parallel for schedule(dynamic)
288     for (int pi = 0; pi < packets_per_row*packets_per_row; ++pi) {
289         ray rays[packet_width*packet_width];
290         int y = pi/packets_per_row*packet_width;
291         int x = pi%packets_per_row*packet_width;
292         for (int py = y, r = 0; py < y + packet_width; ++py)
293             for (int px = x; px < x + packet_width; ++px, ++r)
294                 set_ray(rays[r], FLT_MAX, eye, c + u*px + v*py);
295         intersect(rays, packet_width*packet_width, 0);
296         shade(rays, packet_width*packet_width);
297         for (int py = y, r = 0; py < y + packet_width; ++py)

```

```

298         for (int px = x; px < x + packet_width; ++px, ++r)
299             image[py*image_size + px] = rays[r].color;
300     }
301     glDrawPixels(image_size, image_size, GL_RGB, GL_FLOAT, image);
302     glFlush();
303 }

```

The `render()` function simulates a pinhole camera, controlled by the eye point, `look_at` point, and field-of-view angle (`fov`) variables. Using these, it essentially sets up an orthogonal basis that maps pixel positions in the image to ray directions.

The image itself is square with `image_size` pixels to a side. In turn, the image is subdivided into a much finer group of set of squares, `packet_width` pixels to a side. For each of these small squares, `render()` assembles a small group, or “packet,” of rays [100, 101] which it sends to the `intersect()` and `shade()` phases. Figure 2.3 shows how the packet is assembled.

Processing rays in packets like this allows an interactive ray tracer to reduce overhead, improve cache efficiency, and exploit instruction level parallelism. For

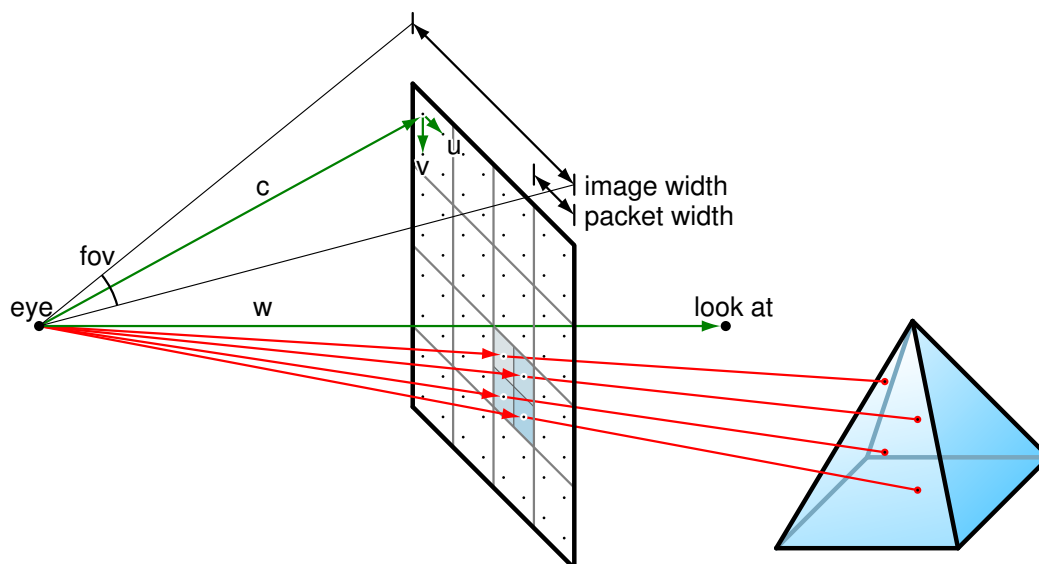


Figure 2.3. Pinhole camera model and assembly of a ray packet.

example, the `ray_tri_intersect` routine only needs to recompute a triangle's normal once per packet rather than once per ray. If the packets have other properties such as a common origin or common direction, even greater benefits are possible, as discussed in Chapter 3. More sophisticated ray tracers can also cull entire packets against nodes in the acceleration structure [83, 11], greatly reducing the incremental cost to adding additional rays to a packet.

Packets also improve cache efficiency by amortizing the cost of a cache miss over all of the rays in the packet. Provided that the rays in the packet are “coherent,” if any of the rays needs to use an acceleration structure node or scene primitive, there's a good chance that other rays in the packet will too. After the first ray in the group touches an acceleration structure node, or a scene primitive, it will be available in the cache for the rest of the group to use.

While Mirth uses simple scalar loops to process each ray in its packets, more sophisticated systems take advantage of modern processor's vector instruction sets such as SSE or AltiVec to perform the computations on the rays with fine-grained instruction level parallelism.

Most interactive ray tracers also use coarser, thread level parallelism where each rendering thread processes packets as units of work. Ray tracing is considered “embarrassingly parallel”; normally no communication between threads is necessary other than assigning packets and synchronizing between frames. This allows it to scale efficiently with multiple cores. In this case, Mirth uses OpenMP [70] with dynamic scheduling to handle threading and the distribution of work.

The last bit of code in Mirth just handles user interaction, sets up OpenGL and GLUT and puts everything into action:

```

305 void reshape(int const width, int const height) {
306     glViewport(0, 0, image_size, image_size);
307 }
308
309 void mouse(int const button, int const state,
310            int const x, int const y) {
311     if (state == GLUT_DOWN && button == GLUT_LEFT_BUTTON)

```

```

312     buttons |= 1;
313     else if (state == GLUT_DOWN && button == GLUT_RIGHT_BUTTON)
314         buttons |= 2;
315     if (state == GLUT_UP && button == GLUT_LEFT_BUTTON)
316         buttons &= ~1;
317     else if (state == GLUT_UP && button == GLUT_RIGHT_BUTTON)
318         buttons &= ~2;
319     mouse_x = x;
320     mouse_y = y;
321 }
322
323 void motion(int const x, int const y) {
324     if (buttons & 1) {
325         yaw += (x - mouse_x)*0.01f;
326         pitch += (y - mouse_y)*0.01f;
327     } else if (buttons & 2)
328         track += (y - mouse_y)*0.01f;
329     eye = vec(cos(pitch)*sin(yaw),
330             cos(pitch)*cos(yaw),
331             sin(pitch))*track + look_at;
332     mouse_x = x;
333     mouse_y = y;
334 }
335
336 int main(int argc, char **argv) {
337     glutInit(&argc, argv);
338     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
339     glutInitWindowSize(image_size, image_size);
340     glutCreateWindow("Mirth");
341     glutIdleFunc(glutPostRedisplay);
342     glutDisplayFunc(render);
343     glutReshapeFunc(reshape);
344     glutMouseFunc(mouse);
345     glutMotionFunc(motion);
346     int num_tris = read_model(argv[1]);
347     bvh = new node[num_tris*2];
348     build_bvh(0, 1, 0, num_tris);
349     image = new vec[image_size*image_size];
350     motion(0, 0);
351     glutMainLoop();
352 }

```

2.2 Noninteractive Ray Tracing

The initial efforts to apply ray tracing to visual synthesis produced systems were far too slow for interactive use. The first of these was by Appel [5], who used it to add half toning to wireframe drawings of polygonal surfaces produced

with a plotter. His technique sampled the image plane along a half tone pattern, tracing rays to determine the hit point on the closest polygon and then tracing an additional ray to determine whether the point was in shadow. Based on this, the plotter drew a plus mark of variable size, depending on the amount of light received at the hit point.

Goldstein and Nagel [33] eliminated the wireframe and relied purely on ray tracing for hidden surface removal. Their system extended ray tracing beyond polygons to other primitives such as spheres, cylinders and cones. These primitives could be combined into more complex objects through combinatorial solid geometry (CSG). Instead of using a plotter, a high resolution grey scale raster CRT displayed the computed images in the frame.

Kay and Greenberg [45] implemented refraction for transparent surfaces through an early hybrid method that combined rasterization with simple ray tracing. Their method rasterized the surfaces in the scene in order from back to front. When rendering a transparent surface the technique used ray tracing and projection to determine the image space offset due to refraction, and mapped the current color from the offset point in the image buffer to pixel being rendered.

Whitted [103] demonstrated the first fully general rendering algorithm based on recursive ray tracing with global scene information. Instead of simply tracing a primary ray into the scene and terminating when the nearest intersection was found, his algorithm could trace additional rays starting from the intersection point to determine light obstructions for shadows, specular reflection and specular refraction. Specular rays could, in turn, cause additional shadow and specular rays to be traced. With this, he introduced the concept of the ray tree, a set of rays that contribute illumination to a single sample.

Cook et al. [20] introduced distribution ray tracing by extending Whitted's ray tracing algorithm with Monte Carlo techniques. By stochastically sampling in time and space according to various distributions they were able to extend ray tracing to incorporate the effects of motion blur, depth of field, soft shadows, translucency and glossy reflections. More importantly, distribution ray tracing

could combine all of these effects quite cleanly.

Kajiya [44] further extended this idea with the introduction of path tracing. This technique collapsed the ray tree by probabilistically following a single path through it from the eye point to the light sources. By continuing to follow light paths that bounced off of diffuse surfaces path tracing could compute diffuse interreflections in addition to the effects made possible with distribution ray tracing. This made path tracing a rather general technique for generating images that account for most of the global illumination within a scene. Its primary limitations are firstly that it is normally quite slow, requiring many samples per pixel to converge to an image that is free from the appearance of noise. Secondly, it can only account for effects of geometric optics; polarization and diffraction are still ignored. Related techniques such as bidirectional ray tracing [53], Metropolis light transport [93], and adjoint photons [64] can improve the statistical efficiency of path tracing, but do not increase the range of images that it can render.

2.3 Interactive Ray Tracing

As the speed and realism of ray tracing-based rendering systems increased, attention also turned towards making it interactive. One early attempt at this was made by Muuss [67] in 1992, who exploited the embarrassingly parallel nature of ray tracing to distribute the rendering of each frame across a single 12-CPU node of an SGI Power Challenge Array supercomputer. This allowed him to achieve up to 1.49 fps for a small CSG scene with six solids in it, and 0.61 fps for a scene with 1126 solids. While borderline interactive, these experiments suggested that truly interactive ray tracing would soon be possible – at least on expensive supercomputers.

Three years later, Parker et al. [73] created a truly interactive ray tracing system. Through careful implementation of a brute-force ray tracing system, they were able to achieve approximately 10 fps while rendering isosurfaces of the full resolution (1GB) Visible Woman dataset on an SGI Reality Monster. This

system directly intersected rays against the isosurface and took advantage of the superior scaling of ray tracing on very large data sets to outperform what would have been possible through conventional polygonal isosurface extraction and rasterization at the time.

Subsequent work by Parker's group [72] extended their system into a more general, full-featured Whitted-style ray tracer, called *-Ray or RTRT. This extended system supported a much larger set of primitives including NURBS surfaces, quadrics and polygonal meshes. It could also render using a variety of material that included diffuse, metal, dielectric and coupled models. This system could render 512×512 images of a dataset consisting of 35 million spheres at approximately 15 fps, though this occupied 60 CPUs of an SGI Origin 2000.

While these systems succeeded at realizing interactive ray tracing, they required large, expensive shared memory supercomputers. Subsequent efforts began to look at making interactive ray tracing possible with inexpensive clusters of workstations. Wald's group [100, 99] produced a distributed system that subdivided the scene database (including geometry, acceleration structure, and shading data) into pieces called voxels. Each cluster node explicitly requested voxels from the central server as needed and managed its own cache of them. In order to hide latency, a node suspended rays that induced voxel request and continued processing other rays until the data arrived.

The DIRT project from DeMarle et al. [22] extended Parker's RTRT to also run on clusters, allowing for the visualization of volume datasets too large to fit within the memory of a single workstation. Where Wald's system used a central server to service request for voxels, DIRT divided the scene data evenly among the cluster nodes and used a peer-to-peer style system to pass scene data between them. Unlike the explicit cache management used by Wald's system, DIRT added a software distributed shared memory system to transparently generate and process the request for blocks of data. This automatic approach sacrificed the ability to hide the latency for these requests, however.

Wald's group [101] had also demonstrated that with careful attention to ray coherence, even a single workstation could ray trace at interactive frame rates for some scenes. Combining small groups of rays into "packets" allowed such a ray tracer to exploit the instruction level parallelism offered by Intel's Streaming SIMD Extensions (SSE) and to improve cache use by amortizing the cost of cache misses over multiple rays. Thus, ray coherence could lead to more efficient use of hardware resources.

Dmitriev et al. [24] then showed that coherence could also lead to improvements in algorithmic efficiency. In particular, a tight "beam" or frustum bounding the rays in a packet could be used as a proxy for rapid rejection tests when intersecting a ray with a triangle. If the frustum for a set of rays falls entirely to beyond one of the triangle's edges, then there is no need to continue testing individual rays in the proxied group against the triangle. While ray tracing has traditionally been considered to scale sublinearly with the size of the scene and linearly with the size of the display, ray coherence could allow for sublinear scaling with the size of the display or the density of the samples as well.

Reshetov et al. [83] extended this idea to acceleration structure traversal. They achieved a very fast kd-tree [9] traversal algorithm for coherent ray packets by the culling branches of the kd-tree that fell outside of the ray frusta. Because the test between the kd-tree node and the ray frusta can be computed in constant time, this leads to a system that scales very well.

Wald et al. [97] used a similar idea for traversing bounding volume hierarchies [84]. Instead of bounding the rays in the packet with frusta, they computed intervals for each component of the ray's origins and directions and then used interval arithmetic [11] to perform the culling test. The effect is the same, however, in that the traversal algorithm can cull complete sub branches without having to check each individual ray against the node.

The third major acceleration structure, uniform grids [30], have also seen significant benefit from coherent ray tracing. Wald et al. [98] developed an algorithm that used the frustum bounding a packet to choose which cell to

traverse, and checked the primitives in those cells against all rays in the packet. With efficient implementation [101] of mail boxing [50], plus Dmitriev's SIMD triangle culling, they were able to see significant gains over tracing single rays at a time when the rays were coherent.

2.4 Custom Hardware for Interactive Ray Tracing

Custom hardware for interactive ray tracing is somewhat more rare. Schmitler et al. [85, 86] proposed the first major design, the SaarCOR, which led to the Ray Processing Unit (RPU) [105]. The first, SaarCOR, used fixed function hardware. The RPU lifted this restriction and offered programmable material, geometry and lighting computations while maintaining dedicated function units for acceleration structure traversal. The RPU offered four-way vector arithmetic, similar to Intel's SSE, and processed each ray in its own thread. Threads were grouped together into synchronous "chunks" and executed in lock step in SIMD fashion. Multiple chunks, however, could be processed independently.

Fender and Rose [28] also developed a fix-function system for an FPGA. Their system used triangles as the primitive, with an unusual three-level BVH of oriented bounding boxes as the acceleration structure. Their system does not appear to have used any kind of threading, though the hardware ray-triangle intersector could intersect one triangle with three rays every three cycles.

The Copernicus architecture [34] (designed to run the Razor software [23]) is completely programmable and offers a four-wide SIMD instruction set based on SSE. Each in-order core would run two to eight threads, and eight of these cores would form a tile with a shared L2. Govindaraju et al. estimate that 16 of the tiles could fit on a 240 mm² chip in a 22 nm technology and reach 74 million rays per second when run at 4GHz.

The TRaX [91] architecture takes a somewhat different approach. It completely dispenses with any sort of SIMD and instead relies completely on efficient execution of large numbers of threads. By reverting to a single-ray programming model, it loses some of the speed benefits offered by ray packets

for primary rays. However, it makes up for this by handling secondary rays nearly as efficiently as primary rays – something that packet and SIMD style ray tracers struggle with [52]. Spjut et al. estimate that the TRaX architecture could reach 50 million rays per second with a 500 MHz 200 mm² chip in a 130 nm process and 177 million rays per second with a 65 nm process.

In the commercial realm, Caustic Graphics has recently demonstrated a ray tracing accelerator [16]. Relatively little about the architecture has been revealed other than that it currently relies on the CPU to perform the shading. It currently exist in FPGA form.

Intel's Larrabee architecture, though not specifically designed for ray tracing has nonetheless been designed with interactive ray tracing in mind as an important application [87]. Larrabee uses a set of in-order x86 cores connected via a ring network to each other and to a set of fixed function units to texture sampling. Rather than the standard 4-wide SSE, however, Larrabee's cores use extended 16-wide vector instructions. Seiler et al. estimate that with 32 cores running at 1GHz, Larrabee could achieve 70 frames per second on a scene requiring four million rays per frame.

CHAPTER 3

OPTIMIZING RAY-TRIANGLE INTERSECTION VIA AUTOMATED SEARCH

This work is based on an earlier work © 2006 IEEE. Reprinted, with permission, from Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, Optimizing Ray-Triangle Intersection via Automated Search, Andrew Kensler and Peter Shirley.

In this chapter, we examine existing direct 3D ray-triangle intersection tests (i.e., those that do not first do a ray-plane test followed by a 2D test) for ray tracing triangles and show how the majority of them are mathematically equivalent. We then use these equivalencies to attempt faster intersection tests for single rays, ray packets with common origins, and general ray packets. We use two approaches, the first of which counts operations, and the second of which uses benchmarking on various processors as the fitness function of an optimization procedure. Combining the approaches by using the operation-counting method to further optimize the code produced via the fitness function produces an efficient ray-triangle test for ray packets.

Following that, we turn from designing a ray-triangle test for pure speed to designing one for numerical robustness. We begin with an exploration of how the size and distribution of bits in a floating point format affects the computations in direct intersection algorithms. Based on this, we design an extremely stable ray-triangle intersection algorithm and show how it produces renderings with fewer artifacts at very lower floating point precisions.

3.1 Introduction

Ray-object intersection is one of the kernel operations in any ray tracer [103], and a different function is implemented for each type of geometric primitive. Triangles are one of the most ubiquitous rendering primitives in use. They typically find use as a “lowest common denominator” between modelers and renderers, due to their simplicity, uniformity and the ease of tessellating more complex primitives into triangles. Many renderers even use triangles as their sole primitives for these reasons. Thus, high performance when rendering triangles is a key feature in nearly every renderer.

There are three basic classes of ray-triangle tests commonly in use (see [59] for a thorough list and empirical comparison for single ray tests). The first intersects the ray with the plane containing the triangle, and then does a 2D point-in-triangle test in the plane of the triangle (e.g., [95, 8]). The second does a direct 3D test based on some algebraic or geometric observation such as provided in Cramer’s rule (e.g., Möller-Trumbore [63]), matrix inversion [6]), ratios of tetrahedral volumes [71], triple products, ratios of determinants, or Plücker coordinates (e.g., [2]). A third class that is uncommon today recursively subdivides the triangle to determine if the ray hits the triangle and where [94]. Dammertz and Keller [21] used a recursive subdivision scheme to compute numerically robust intersection. This chapter examines only the direct 3D test, and observes that “under the hood” these methods are all taking ratios of volumes, and differ mainly in what volumes are computed.

For these 3D methods we optimize ray-triangle intersection in two different ways. First we do explicit operation counting for the cases of single rays, packets of rays with common origins, and general packets of rays. Next we do code evaluation by letting a genetic algorithm modify the code using profiling on various computers as a fitness function. The implementation is based on SIMD and ray packets to improve the chances of relevance for modern implementations.

3.2 Signed Volumes

The signed area of the parallelogram shown in the left of Figure 3.1 is given by

$$\text{area} = \begin{vmatrix} x_a & x_b \\ y_a & y_b \end{vmatrix}.$$

If we were to switch \mathbf{a} and \mathbf{b} , the sign would change. The sign is positive when the second vector is in the counterclockwise direction from the first. There is a similar signed volume rule for parallelepipeds such as the one shown in the right of Figure 3.1:

$$\text{volume} = \begin{vmatrix} x_a & x_b & x_c \\ y_a & y_b & y_c \\ z_a & z_b & z_c \end{vmatrix}.$$

This volume is positive if the vectors form a right-handed basis, and negative otherwise. The volume of the tetrahedron defined by the three vectors is one-sixth that of the parallelepiped's.

The volume formula can be used to compute 2D triangle area by embedding the triangle in 3D with the three vertices on the $z = 1$ plane as shown in Figure 3.2:

$$\text{triangle area} = \frac{1}{2} \begin{vmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{vmatrix}. \quad (3.1)$$

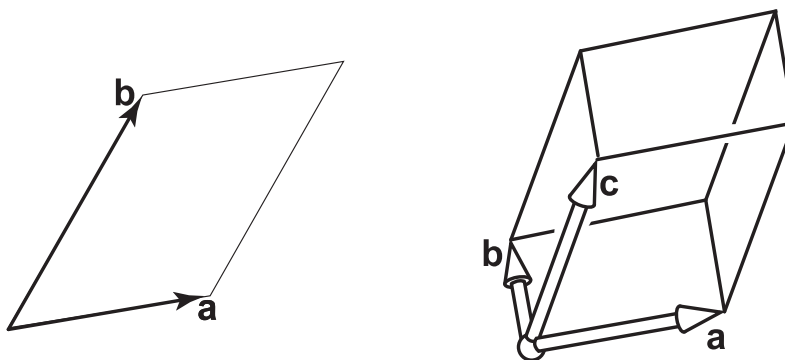


Figure 3.1. Determinants and signed areas or volumes. The signed area or volume of these objects are given by determinants with the Cartesian coordinates of the vectors as matrix rows or columns. The sign of each of these examples is positive via right hand rules.

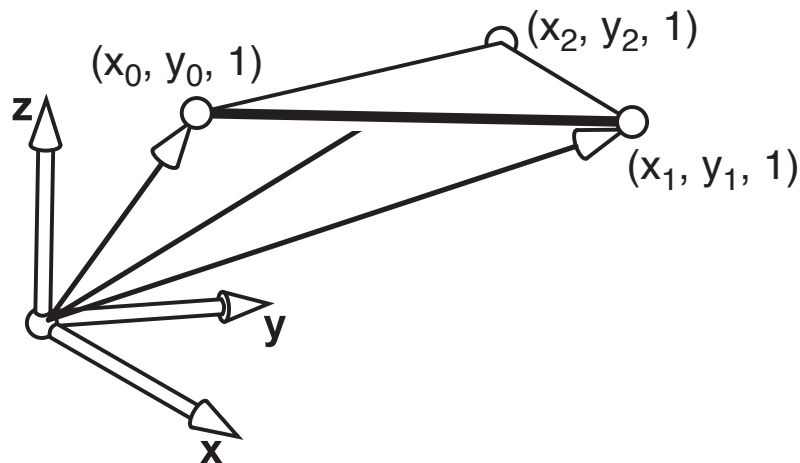


Figure 3.2. Homegenous coordinates and triangle area. The area of the triangle can be computed from the volume of the parallelepiped determinants with the Cartesian coordinates of the vectors as matrix rows or columns. The sign of each of this example is positive via right-hand rules.

The reason for the first one-half is that the area of the triangle is three times the volume of the tetrahedron defined by the three column vectors in the matrix, and the determinant is six times the volume of that tetrahedron. We can also use the determinant rule to observe:

$$\text{triangle area} = \frac{1}{2} \begin{vmatrix} x_1 - x_0 & x_2 - x_0 \\ y_1 - y_0 & y_2 - y_0 \end{vmatrix}.$$

This second (2D) determinant is the area of the parallelogram defined by the two 2D edge vectors of the triangle, and has the same value as the determinant in Equation 3.1, although this is not algebraically obvious. This is an example of why interpreting determinants as area/volume computations can be better, especially for geometric thinkers.

The volume of a tetrahedron defined by four vertices \mathbf{p}_i can be found by taking the determinant of three of the vectors along its edges, or by a 4D determinant on a $w = 1$ plane:

$$\text{volume} = -\frac{1}{6} \begin{vmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \\ 1 & 1 & 1 & 1 \end{vmatrix} = \frac{1}{6} \begin{vmatrix} x_1 - x_0 & x_2 - x_0 & x_3 - x_0 \\ y_1 - y_0 & y_2 - y_0 & y_3 - y_0 \\ z_1 - z_0 & z_2 - z_0 & z_3 - z_0 \end{vmatrix}.$$

The minus sign before the first determinant is not a mistake. Some care must be taken on the ordering rules for different matrix forms in the various dimensions; the odd dimensions have a sign change between the edge-vector based method and the $w = 1$ hypervolume method.

There are two other ways by which signed volumes are often computed in 3D. The first is the triple product (equivalent to a determinant in 3D):

$$\text{volume} = \frac{1}{6} [(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)] \cdot (\mathbf{p}_3 - \mathbf{p}_0).$$

Another method for computing a signed volume uses the Plücker inner product for the directed line segments $\mathbf{p}_0\mathbf{p}_1$ and $\mathbf{p}_2\mathbf{p}_3$. This is algebraically the same as the determinant and triple product methods [43].

3.3 Using Signed Volumes For Intersection

The basic signed volume idea has been used by several researchers for ray-triangle intersection, and the equivalence between Plücker inner products, triple products, and determinants for intersection has been pointed out by O'Rourke [71]. For example, consider the configuration in Figure 3.3. The signed volume of the tetrahedron $\mathbf{a}\mathbf{p}_2\mathbf{p}_0$ is given by:

$$V_1 = \frac{1}{6} [(\mathbf{p}_2 - \mathbf{a}) \times (\mathbf{p}_0 - \mathbf{a})] \cdot (\mathbf{b} - \mathbf{a}).$$

If this sign is negative, then the ray is to the "inside" side of the edge. The magnitude of V_1 is proportional to the area of the shaded triangle. Similarly, areas V_0 and V_2 can be computed with respect to the edges opposite \mathbf{p}_0 and \mathbf{p}_2 (see Figure 3.4). If all three V_i are the same sign, then the infinite line through \mathbf{a} and \mathbf{b} hits the triangle. The barycentric coordinates can also be recovered. For example:

$$\alpha = \frac{V_0}{V_0 + V_1 + V_2}, \quad \beta = \frac{V_1}{V_0 + V_1 + V_2}.$$

The segment hits the triangle if the signed volume of the tetrahedron $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\mathbf{a}$ and $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2\mathbf{b}$ have the opposite signs. If these volumes are V_a and V_b , and V_a is positive, then the ray parameter is given by:

$$t = \frac{V_a}{V_a - V_b}.$$

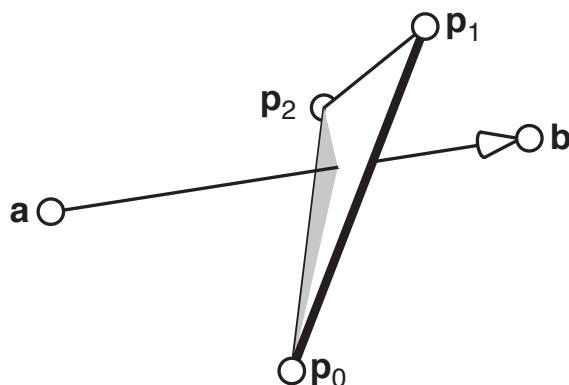


Figure 3.3. Geometry for a ray edge test.

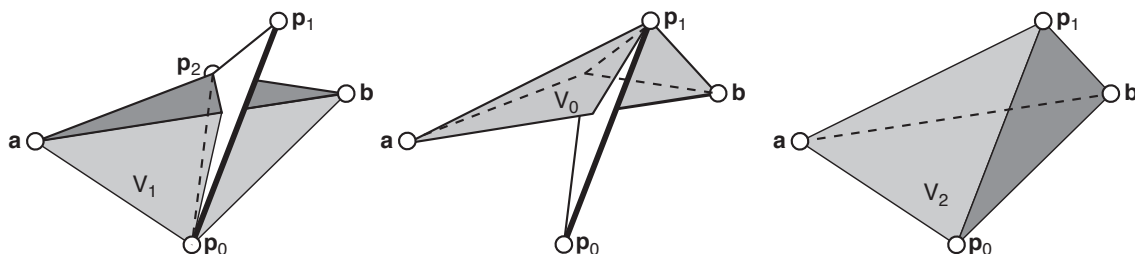


Figure 3.4. Volumes for all ray edge tests.

Note that you could also compute the volume of $V_0 + V_1 + V_2$ directly:

$$V = V_0 + V_1 + V_2 = \frac{1}{6} [(\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)] \cdot (\mathbf{b} - \mathbf{a}).$$

Note also that that is the denominator in Cramer's Rule test, which under-the-hood is computing volumes.

If volumes are to be used, there are several degrees of freedom which can be exploited to yield different tests. For example, one can compute the inside/outside test for the whole ray in several ways:

1. compute V_0, V_1, V_2 , test for same sign;
2. compute $\alpha = V_0/V, \beta = V_1/V, \gamma = 1 - \alpha - \beta$, test for all in $[0, 1]$.
3. compute $\alpha = V_0/V, \beta = V_1/V, \gamma = V_2/V$, test for all in $[0, 1]$.

In addition, each of the volumes can be computed via several different edge tests. For example, the volume V_1 has six edges, any of which can be followed in either direction. Any three edges that are not all mutually coplanar will yield the same volume, though possibly with the opposite sign. For a volume defined by four points, there are 384 unique ways to compute the same signed volume. Given three points and a direction vector, there are 36 ways to compute the same signed volume. The one above allows a ray packet to precompute the cross product if the ray origin is shared. This may or may not be useful for sharing computations (i.e., subexpressions).

For the ray parameter test, $V = V_a - V_b$ is in fact the same V as above. Overall, a test must directly compute at least two of V_0, V_1, V_2 , and at least one of V_a and V_b . Finally one of the remaining three volumes must be computed directly.

3.4 Minimizing Total Operation Counts

In this section we try to use the equations that minimize the total number of operations. Because of the large number of possible equations, we use a brute force searching method to examine all cases. In the next section we use a more sophisticated and empirical method to optimize performance on real processors.

Volume-based triangle tests require the computation of at least four volumes for a successful intersection. These are generally either one of V_0 or V , plus V_1, V_2 and V_a . The exhaustive search considered every possible set of four scalar triple products to compute these volumes and for each of these sets, the total number of floating point operations, taking into account common subexpressions.

For possibilities, our program makes a list of the cost in number of arithmetic operations associated with each subexpression. For the example expression $c = ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot (\mathbf{p}_0 - \mathbf{a}), (\mathbf{p}_1 - \mathbf{p}_0), (\mathbf{p}_0 - \mathbf{p}_2), (\mathbf{p}_0 - \mathbf{a})$ count as three subtractions each. $((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2))$ counts as six multiplies and three subtractions (since $\mathbf{p}_1 - \mathbf{p}_0$ and $\mathbf{p}_0 - \mathbf{p}_2$ will already have been counted). The

whole expression for c costs three multiplies and two additions since, again, the subexpressions for the dot product were counted elsewhere.

With these lists in hand, the program looks at every combination of scalar triple products for the volumes. For example, if the ray stores $\mathbf{d} = \mathbf{b} - \mathbf{a}$, it would examine:

$$\begin{aligned}
 V &= (\mathbf{d} \times (\mathbf{p}_2 - \mathbf{p}_0)) \cdot (\mathbf{p}_1 - \mathbf{p}_0) \\
 V_a &= ((\mathbf{a} - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)) \cdot (\mathbf{p}_2 - \mathbf{p}_0) \\
 V_1 &= (\mathbf{d} \times (\mathbf{p}_2 - \mathbf{p}_0)) \cdot (\mathbf{a} - \mathbf{p}_0) \\
 V_2 &= ((\mathbf{a} - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_0)) \cdot \mathbf{d}.
 \end{aligned} \tag{3.2}$$

while checking every combination of scalar triple products for calculating each of the four volumes (i.e., $36 \cdot 384 \cdot 36 \cdot 36$ possibilities).

So for each of these combinations of expressions, it counts the number of operations, but duplicate operations are only counted once each. With the example above, it would count $(\mathbf{p}_2 - \mathbf{p}_0)$ for three subtractions the first time it appeared but not the two subsequent times and $\mathbf{d} \times (\mathbf{p}_2 - \mathbf{p}_0)$ would count only once (not twice) for six multiplies and three subtractions, etc. Given this, it counts up a total number of arithmetic operations under the assumption that all subexpressions are computed once and then the results are saved. The result of this is a list of sets of each set of expressions for the volumes with the lowest cost found.

The program also had a few switches to consider different cases. These mainly affected how the cost was computed. For example, for packets, any subexpression that does not involve \mathbf{d} or \mathbf{a} is independent of the ray, and counts at 1/64th the normal cost (i.e., as though it were amortized over an 8x8 packet.) The exact divisor does not matter hugely since the total flops in the best expressions sets already total well below 64. The sum of these amortized computations in these best cases never totals above 1.0, which means that it will not cause it to beat out cases where it does not choose to amortize. But the fraction does serve

as a tie-breaker to get it to minimize the amount of per-triangle precomputation that it does.

For the general case of single rays and choosing to find V instead of V_0 , there were six optimal formulations requiring a total of 47 operations. Three of these were symmetric cases of the other three with triangle edges reversed and appropriate adjustments made to preserve signs. One of these formulations corresponds to the Möller-Trumbore algorithm [63], and was already given in Equations 3.2. When V_0 is used instead of V , the case is similar and there are still just six analogous best formulations, each requiring 47 operations.

With ray packets, however, all computations involving only the triangle vertices can be amortized over all of the rays in the packet. Assuming that the number of rays in the packet is large enough that all computations that can be amortized over the packet are essentially “free” (though not with zero cost), and that we again choose to use V instead of V_0 , there were exactly two optimal formulations, each symmetric with the other:

$$\begin{aligned} V &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot \mathbf{d} \\ V_a &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot (\mathbf{p}_0 - \mathbf{a}) \\ V_1 &= ((\mathbf{p}_0 - \mathbf{a}) \times \mathbf{d}) \cdot (\mathbf{p}_0 - \mathbf{p}_2) \\ V_2 &= ((\mathbf{p}_0 - \mathbf{a}) \times \mathbf{d}) \cdot (\mathbf{p}_1 - \mathbf{p}_0). \end{aligned}$$

This formulation requires just over 32 operations per ray plus the amortized per-triangle operations. Note that the per-triangle computation simply involves finding two edges plus the unscaled normal of the triangle.

If all of the rays in the packet share a common origin, as is typical for primary rays and shadow rays for point light sources, it is possible to do far better yet. For these cases, all computations involving only the triangle vertices and \mathbf{a} are amortized over the packet. There are 12 optimal formulations (six being symmetrical with the other six), and requiring just over 15 operations per ray to find the volumes in the inner loop. At this point, V_a may be amortized entirely as well as all of the cross products, leaving only the three dot products:

$$\begin{aligned}
V &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot \mathbf{d} \\
V_a &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{p}_2)) \cdot (\mathbf{p}_0 - \mathbf{a}) \\
V_1 &= ((\mathbf{p}_0 - \mathbf{p}_2) \times (\mathbf{p}_0 - \mathbf{a})) \cdot \mathbf{d} \\
V_2 &= ((\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_0 - \mathbf{a})) \cdot \mathbf{d}.
\end{aligned}$$

When V_0 is used instead of V , the case is similar and there are still just 12 analogous best formulations, each requiring 15 operations. This property has been used to advantage by Benthin in his dissertation, although he derived it through Plücker coordinates. [8]

3.5 A Genetic Algorithm for Improved Performance

While operation counts are correlated to performance, the increasingly complex hardware and compiler technology makes optimization largely an empirical process. Since the number of possibilities is so large for how code can be written, exhaustive search by hand is not a good option. In this section we use genetic algorithms to improve our choices among coding options in a spirit similar to that shown effective for sorting [58].

Before applying any genetic search, we first formalize the space of choices we have. For example, we can compute V_0 , V_1 , and V_2 and derive V , or we could compute V_0 , V_1 and V derive V_2 . Another option is whether to test for early exit if a given variable is outside its allowed range. This test must of course come after that variable is computed. On the other hand, some quantities can be computed in any order. This suggests a dependency graph.

For the ray-triangle intersection tests, we used a dependency graph with 1294 nodes. The allowed parameter space included all legitimate choices for the signed-volume computations for the t-value, V , V_0 , V_1 , and V_2 , the choice between computing V directly with a single signed volume computation or by summation of the three, how long to postpone the division, whether to use a barycentric in/out test or to test in/outness by comparing the signs of the volumes, whether and where to use early exits if all four rays in an SSE bundle are known to miss, etc.

The genetic algorithm approach used to sort among these options consists of three parts: the main genetic algorithm driver, the benchmark, the code generator.

The main genetic algorithm implementation is an evolution algorithm very similar to that used by Li et al. [58]. In this, the best individuals in each generation survive to the next generation entirely unchanged. Genetic recombination applies only to creating the new offspring to replace the least fit individuals. These are also subject to occasional random mutations. As with their system, we used a population of 50 individuals through 100 generations. At each generation, the 20 most fit were kept unchanged while the 30 least fit were replaced with pairs of offspring created through recombination from two parents with a two-point crossover from the parent generation (a random middle segment from one parent is replaced with those values in the order that they appear in the other parent, and vice versa, to produce a pair of offspring that are still permutations. The reason that genomes must be permutations is due to the way they control code emission and is explained later.) The parents were chosen with probability proportional to their fitness. After this, two mutations were applied to the offspring at random, by swapping a random pair of values in their genetic sequence.

After this, the new generation is evaluated for fitness, which in this case consists of using each genome to output code for a ray-triangle intersection test combined with a benchmark for speed. The created program consists of a fixed, handwritten template for the benchmark with the genetically derived intersection test inserted into the template. This is compiled and executed, and the measured speed in millions of intersections per second becomes the fitness value for that individual. The benchmark code measures the performance of 20000 random triangles each intersected by 400 packets of 64 random rays each. The positions of the rays and triangles are chosen such that the intersection probability is approximately 25%. This probability was chosen to mimic the case for a good acceleration structure where rays that reach the point of intersecting

a triangle have a high probability of success. Fifty percent is a best case for this due to the typical tessellation of quads into pairs of triangles, where each triangle in the pair will typically have significantly overlapping bounding boxes but only a 50% or so chance of hitting, once a ray reaches the bounding box.

The code generation from the genomes is the most complex part of our process and is inspired by the approach in Fang et al. [26]. Each individual's genome encodes the algorithm as a permutation of the first 1294 natural numbers. A DAG of dependencies, starting with a goal node gives the list of possible code chunks to output (generally at the level of a single scalar or vector operation) along with any dependencies that must be satisfied before the chunk can be output. These dependencies take two forms: "required" and "optional" dependencies. For each chunk, all required dependencies must be satisfied before the a statement can be output, while only one or more of the optional dependencies needs to be. This distinction means that any generated program that satisfies this dependency graph will have the freedom to choose from alternative code paths where necessary, but will also be constrained to always generate legal programs that will compile and execute correctly. For example, computing a barycentric coordinate may be done through any of the numerous choices for computing the signed volume, but an early exit test based on the coordinate always requires the coordinate computation as a prerequisite.

Output from this dependency graph is guided by each individuals genome. The genome, as a permutation of the whole numbers, gives the priority for each node in the dependency graph. Code is emitted by applying a modified topological sort to the dependency graph where ties for which statement to emit next are broken according to the priority given in the genome. If an optional dependency has not already been satisfied due to another node, the optional dependency with the highest priority is chosen. An initial depth-first walk of the dependency graph from the goal node marks live nodes, so that only these are considered for output during the topological sort. Thus, so long as each genome remains a proper permutation of the first n natural numbers, where n

in this case is 1294 – the number of nodes in the dependency graph – the code generator will always emit a valid and nearly minimal code sequence for it. The genetic algorithm still has tremendous freedom in choosing the relative order of the statements, and through careful encoding in the dependency graph nearly any choices for valid code may be given to the genetic algorithm.

We ran the genetic algorithm both for general and common origin packets. We implemented the code in C++ with SSE extensions. The best program for both packet conditions was then hand optimized making minor performance improvements.

The hand tuning was quite minimal. We examined the code from the genetic algorithm to determine what choices it made for how to compute the signed volumes. Then, we examined the list of optimal operation-count expressions from the exhaustive search in the previous section and found the most similar set of expressions to that from the genetic algorithm. We then changed the code from the genetic algorithm to use the expressions from the optimal search, trying to change the code and especially the basic structure as little as possible. Typically this involved reversing the direction of an edge here and changing the operands for a dot or cross product there. Next we cleaned up the dead code left over from the previous step, since taking better advantage of common subexpressions meant that some of the former computations were now extraneous. Lastly, we cleaned up the artifacts from the genetic algorithm – for example, as the final SIMD mask is the result of ANDing the masks from several tests, and these may be done in any order, the mask is initially set to all true before being ANDed with the first test. The obvious optimization, however, is to initialize the mask to the result of the first test. There were one or two similar cases where artifacts from the genetic algorithm could be cleaned up by the compiler’s optimizer. We simply applied the same transformations to streamline source. Overall, the changes we made were quite mechanical and not large.

The code from the genetic algorithm (GA) and the hand-tuned code (tuned GA) were tested against a direct ray packet and SIMD adaptation of Möller-

Trumbore test as indicated in Table 3.1. As can be seen, significant speedups are possible.

3.6 General Packet Code

This annotated code shows our best performing code (derived by hand-tuning the output of the genetic algorithm) for general packets, and shows in detail how we tested its performance.

```

1 #include <mmintrin.h>
2 #include <xmmintrin.h>
3 #include <emmintrin.h>
4 #include <stdlib.h>
5 #include <time.h>
6 #include <sys/time.h>
7 #include <fstream>
8 #include <iostream>
9 using namespace std;
10 static const int packet_size = 64;
11 static const int number_of_packets = 400;
12 static const int number_of_triangles = 20000;
13 static const float eye_range = 3.0f;

```

Table 3.1. Performance comparison of ray-triangle intersection algorithms. Numbers measure indicate millions of rays intersected per second measured on a single core. Compiler target and test platforms are a 2.4GHz Dual Core Opteron (Opt), 3.2GHz Dual Core Xeon (X), 3.0GHz Canterwood Pentium 4 (P4), and a 1.83GHz Core Duo (C).

Compiler	Target	Test	Möller- Trumbore	General Packets		Shared Origin	
				GA	Tuned GA	GA	Tuned GA
GCC 4.02	Opt	Opt	82.0	158.7	164.7	201.2	190.9
GCC 4.02	P4	Opt	71.5	115.5	141.7	158.8	182.1
ICC 9.0	P4	Opt	104.5	135.4	158.8	182.2	203.3
GCC 3.35	X	X	89.0	158.8	173.0	189.3	216.1
ICC 9.0	P4	X	124.7	163.4	180.3	207.3	228.4
GCC 4.02	P4	P4	66.9	97.1	102.6	173.5	158.0
ICC 9.0	P4	P4	115.6	167.8	191.0	205.9	229.3
GCC 4.01	C	C	53.6	81.6	106.3	112.8	125.2
Average			88.5	134.8	152.3	178.9	191.7

```

14 static const float target_range = 0.6f;
15 static const float ray_jitter = 0.04f;
16 // Triangle vertex positions
17 float p0xf[number_of_triangles];
18 float p0yf[number_of_triangles];
19 float p0zf[number_of_triangles];
20 float p1xf[number_of_triangles];
21 float p1yf[number_of_triangles];
22 float p1zf[number_of_triangles];
23 float p2xf[number_of_triangles];
24 float p2yf[number_of_triangles];
25 float p2zf[number_of_triangles];
26 // Ray origins, directions and best t-value
27 float oxf[number_of_packets][packet_size];
28 float oyf[number_of_packets][packet_size];
29 float ozf[number_of_packets][packet_size];
30 float dx[number_of_packets][packet_size];
31 float dyf[number_of_packets][packet_size];
32 float dzf[number_of_packets][packet_size];
33 float rtf[number_of_packets][packet_size];
34 int main(int argc, char **argv) {
35     int seed_time = time(0);
36     unsigned short seeds[] = {
37         static_cast<unsigned short>(seed_time & 0xffff),
38         static_cast<unsigned short>((seed_time >> 8) & 0xffff),
39         static_cast<unsigned short>((seed_time >> 16) & 0xffff) };
40     seed48(seeds);
41     // Setup tests with random triangles and packets
42     for (int ti = 0; ti < number_of_triangles; ++ti) {
43         p0xf[ti] = drand48() - drand48();
44         p0yf[ti] = drand48() - drand48();
45         p0zf[ti] = drand48() - drand48();
46         p1xf[ti] = drand48() - drand48();
47         p1yf[ti] = drand48() - drand48();
48         p1zf[ti] = drand48() - drand48();
49         p2xf[ti] = drand48() - drand48();
50         p2yf[ti] = drand48() - drand48();
51         p2zf[ti] = drand48() - drand48();
52         float mx = (p0xf[ti] + p1xf[ti] + p2xf[ti]) / 3.0f;
53         float my = (p0yf[ti] + p1yf[ti] + p2yf[ti]) / 3.0f;
54         float mz = (p0zf[ti] + p1zf[ti] + p2zf[ti]) / 3.0f;
55         p0xf[ti] -= mx;
56         p0yf[ti] -= my;
57         p0zf[ti] -= mz;
58         p1xf[ti] -= mx;
59         p1yf[ti] -= my;
60         p1zf[ti] -= mz;
61         p2xf[ti] -= mx;
62         p2yf[ti] -= my;
63         p2zf[ti] -= mz;
64     }
65     for (int pi = 0; pi < number_of_packets; ++pi) {

```

```

66  float ex = (drand48() - drand48()) * eye_range;
67  float ey = (drand48() - drand48()) * eye_range;
68  float ez = (drand48() - drand48()) * eye_range;
69  float tx = (drand48() - drand48()) * target_range;
70  float ty = (drand48() - drand48()) * target_range;
71  float tz = (drand48() - drand48()) * target_range;
72  for (int ri = 0; ri < packet_size; ++ri) {
73      oxf[pi][ri] = ex + (drand48() - drand48()) * ray_jitter;
74      oyf[pi][ri] = ey + (drand48() - drand48()) * ray_jitter;
75      ozf[pi][ri] = ez + (drand48() - drand48()) * ray_jitter;
76      dxf[pi][ri] = tx - ex +
77          (drand48() - drand48()) * ray_jitter;
78      dyf[pi][ri] = ty - ey +
79          (drand48() - drand48()) * ray_jitter;
80      dzf[pi][ri] = tz - ez +
81          (drand48() - drand48()) * ray_jitter;
82      rtf[pi][ri] = 1000000.0f;
83  }
84  }
85  timeval start;
86  gettimeofday(&start, 0);
87  // Intersection test begins here
88  for (int pi = 0; pi < number_of_packets; ++pi) {
89      for (int ti = 0; ti < number_of_triangles; ++ti) {
90          // Get triangle corners, compute two edges and normal.
91          // (Alternatively, can precompute and store them)
92          const __m128 p1x = _mm_set_ps1(p1xf[ti]);
93          const __m128 p1y = _mm_set_ps1(p1yf[ti]);
94          const __m128 p1z = _mm_set_ps1(p1zf[ti]);
95          const __m128 p0x = _mm_set_ps1(p0xf[ti]);
96          const __m128 p0y = _mm_set_ps1(p0yf[ti]);
97          const __m128 p0z = _mm_set_ps1(p0zf[ti]);
98          const __m128 edge0x = _mm_sub_ps(p1x, p0x);
99          const __m128 edge0y = _mm_sub_ps(p1y, p0y);
100         const __m128 edge0z = _mm_sub_ps(p1z, p0z);
101         const __m128 p2x = _mm_set_ps1(p2xf[ti]);
102         const __m128 p2y = _mm_set_ps1(p2yf[ti]);
103         const __m128 p2z = _mm_set_ps1(p2zf[ti]);
104         const __m128 edge1x = _mm_sub_ps(p0x, p2x);
105         const __m128 edge1y = _mm_sub_ps(p0y, p2y);
106         const __m128 edge1z = _mm_sub_ps(p0z, p2z);
107         const __m128 normalx = _mm_sub_ps(
108             _mm_mul_ps(edge0y, edge1z),
109             _mm_mul_ps(edge0z, edge1y));
110         const __m128 normaly = _mm_sub_ps(
111             _mm_mul_ps(edge0z, edge1x),
112             _mm_mul_ps(edge0x, edge1z));
113         const __m128 normalz = _mm_sub_ps(
114             _mm_mul_ps(edge0x, edge1y),
115             _mm_mul_ps(edge0y, edge1x));
116         const __m128 zeroes = _mm_setzero_ps();
117         // Loop over "packlets," computing four rays at a time

```

```

118     for (int ri = 0; ri < packet_size; ri += 4) {
119         // Load origin, current t-value and direction
120         const __m128 ox = _mm_load_ps(&oxf[pi][ri]);
121         const __m128 oy = _mm_load_ps(&oyf[pi][ri]);
122         const __m128 oz = _mm_load_ps(&ozf[pi][ri]);
123         const __m128 oldt = _mm_load_ps(&rtf[pi][ri]);
124         const __m128 dx = _mm_load_ps(&dxs[pi][ri]);
125         const __m128 dy = _mm_load_ps(&dys[pi][ri]);
126         const __m128 dz = _mm_load_ps(&dzs[pi][ri]);
127         // Compute volume V, the denominator
128         const __m128 v = _mm_add_ps(_mm_add_ps(
129             _mm_mul_ps(normalx, dx),
130             _mm_mul_ps(normaly, dy)),
131             _mm_mul_ps(normalz, dz));
132         // Reciprocal estimate of V with one round of Newton
133         const __m128 rcpi = _mm_rcp_ps(v);
134         const __m128 rcp = _mm_sub_ps(
135             _mm_add_ps(rcpi, rcpi),
136             _mm_mul_ps(_mm_mul_ps(rcpi, rcpi),
137                 v));
138         // Edge from ray origin to first triangle vertex
139         const __m128 edge2x = _mm_sub_ps(p0x, ox);
140         const __m128 edge2y = _mm_sub_ps(p0y, oy);
141         const __m128 edge2z = _mm_sub_ps(p0z, oz);
142         // Compute volume Va
143         const __m128 va = _mm_add_ps(_mm_add_ps(
144             _mm_mul_ps(normalx, edge2x),
145             _mm_mul_ps(normaly, edge2y)),
146             _mm_mul_ps(normalz, edge2z));
147         // Find Va/V to get t-value
148         const __m128 t = _mm_mul_ps(rcp, va);
149         const __m128 tmaskb = _mm_cmplt_ps(t, oldt);
150         const __m128 tmaska = _mm_cmpgt_ps(t, zeroes);
151         __m128 mask = _mm_and_ps(tmaska, tmaskb);
152         if (_mm_movemask_ps(mask) == 0x0) continue;
153         // Compute the single intermediate cross product
154         const __m128 intermx = _mm_sub_ps(
155             _mm_mul_ps(edge2y, dz),
156             _mm_mul_ps(edge2z, dy));
157         const __m128 intermy = _mm_sub_ps(
158             _mm_mul_ps(edge2z, dx),
159             _mm_mul_ps(edge2x, dz));
160         const __m128 intermz = _mm_sub_ps(
161             _mm_mul_ps(edge2x, dy),
162             _mm_mul_ps(edge2y, dx));
163         // Compute volume V1
164         const __m128 v1 = _mm_add_ps(_mm_add_ps(
165             _mm_mul_ps(intermx, edge1x),
166             _mm_mul_ps(intermy, edge1y)),
167             _mm_mul_ps(intermz, edge1z));
168         // Find V1/V to get barycentric beta
169         const __m128 beta = _mm_mul_ps(rcp, v1);

```

```

170     const __m128 bmask = _mm_cmpge_ps(beta, zeroes);
171     mask = _mm_and_ps(mask, bmask);
172     if (_mm_movemask_ps(mask) == 0x0) continue;
173     // Compute volume V2
174     const __m128 v2 = _mm_add_ps(_mm_add_ps(
175         _mm_mul_ps(intermx, edge0x),
176         _mm_mul_ps(intermy, edge0y)),
177         _mm_mul_ps(intermpz, edge0z));
178     // Test if alpha > 0
179     const __m128 v1plusv2 = _mm_add_ps(v1, v2);
180     const __m128 v12mask = _mm_cmple_ps(
181         _mm_mul_ps(v1plusv2, v),
182         _mm_mul_ps(v, v));
183     // Find V2/V to get barycentric gamma
184     const __m128 gamma = _mm_mul_ps(rcp, v2);
185     const __m128 gmask = _mm_cmpge_ps(gamma, zeroes);
186     mask = _mm_and_ps(mask, v12mask);
187     mask = _mm_and_ps(mask, gmask);
188     if (_mm_movemask_ps(mask) == 0x0) continue;
189     // Update stored t-value for closest hits
190     _mm_store_ps(&rtf[pi][ri],
191         _mm_or_ps(_mm_and_ps(mask, t),
192             _mm_andnot_ps(mask, oldt)));
193     // Optionally store barycentric beta and gamma too
194 }
195 }
196 }
197 // Show speed in millions of intersections per second
198 timeval now;
199 gettimeofday(&now, 0);
200 float elapsed =
201     (static_cast<float>(now.tv_sec - start.tv_sec) +
202     static_cast<float>(now.tv_usec - start.tv_usec) /
203     1000000.0f);
204 if (argc > 1) {
205     ofstream out(argv[1], ios::out);
206     out << (number_of_packets * packet_size
207         * number_of_triangles
208         / elapsed / 1000000);
209 }
210 else
211     cout << (number_of_packets * packet_size
212         * number_of_triangles
213         / elapsed / 1000000) << endl;
214 return 0;
215 }

```

3.7 Numerical Robustness

While raw speed is certainly an important criteria for a ray-triangle intersection algorithm meant for use in an interactive ray tracing, it is not the only consideration. Numerical robustness can also be important. Obviously, a more robust intersection algorithm can improve visual quality by producing fewer artifacts such as surface acne, mesh cracks, or z-fighting. However, it can also have an effect on the speed of a hardware based interactive ray tracer. By choosing a more robust algorithm, we may potentially be able to use smaller, lower precision numbers in the computations. This could lead to smaller, faster functional units performing the calculations, and better cache use by reducing the number of bytes each value takes. An important question, however, is: how many bits in our floating point values do we actually need?

IEEE 754 binary floating point values are represented in the form $(-1)^s \times f \times 2^e$ using three parts, stored together in a bit field: a sign bit, s , a fractional value or significand, f , and an exponent, e . The sign bit indicates that the value is positive if it is clear, and negative if set. The exponent is stored as an unsigned integer, but biased. If eight bits are allocated to the exponent, the value 127 is added to the exponent before storing it and 1023 is added to a 10-bit exponent. The significand normally stores the bits of a fraction between one inclusive, and two exclusive, where the single bit to the left of the radix point is assumed to be a one and is not stored. Each arithmetic operation on floating point numbers normalizes the result to preserve this implicit bit. When the stored exponent is zero, but the significand is nonzero, this represents a subnormal value that is gradually underflowing. If all bits in the exponent are set, it represents a special value; with all bits in the significand clear, it indicates either a positive or negative infinity ($\pm\text{INF}$). However, if any bits in the significand are set, it represents an invalid number, or NaN.

To determine how many bits in the exponent and significand we actually need, and evaluate how reduced precision arithmetic affects ray-triangle intersection, we began by modifying the Manta interactive ray tracer [10] to be able

to produce a trace of all ray-triangle pairs tested for intersection while rendering a frame. This produced files with simple records of 15 single precision floating point values – three coordinates each for \mathbf{a} , \mathbf{d} , \mathbf{p}_0 , \mathbf{p}_1 , and \mathbf{p}_2 . We collected these for path trace renderings of the Happy Buddha, conference room, fairy forest, gargoyle statue, power plant section 16, Soda Hall, and Sponza Atrium. Concatenated together, these provided data for 12,237,461 ray-triangle pairs. Note that these are all pairs that passed initial culling by the BVH acceleration structure. They represent realistic candidates for primitive intersection testing.

Next, we developed a reduced precision library that can simulate IEEE 754 style floating point arithmetic for smaller significand and exponent widths. Internally, this library stores values as double and uses the hardware FPU to compute with them. After each operation, however, it examines the bits in the significand and exponent. The least significant significand bits are forced to zero through rounding, and the exponent is checked to ensure that its value could be represented adequately at the narrower size. If the exponent is too small, additional bits of the significand may be rounded to simulate subnormals, or the value may be forced to zero entirely. If the exponent is too big, the value is set to $\pm\text{INF}$.

Initially, we also developed an corresponding fixed point library to explore how the numbers of bits in the integer and fractional parts affects the ray-triangle intersection tests. However, we quickly found that no combination of field widths totalling to 32 bits (the same size as a single precision floating point number) or fewer was suitable for directly computing the ray-triangle intersections from our test data. Either too few bits were allocated to the fractional part in the case of direction vectors and coordinates for scenes with small scales, or else the integer part overflowed for scenes with larger scales. Figure 3.5 shows the distribution of the numbers in the data by the base-2 exponent in their floating point representation. Every exponent between -33 and 18, inclusive, is represented, for a span of values covering 52 orders in binary magnitude. This span is simply infeasible for a fixed point representation using a machine word

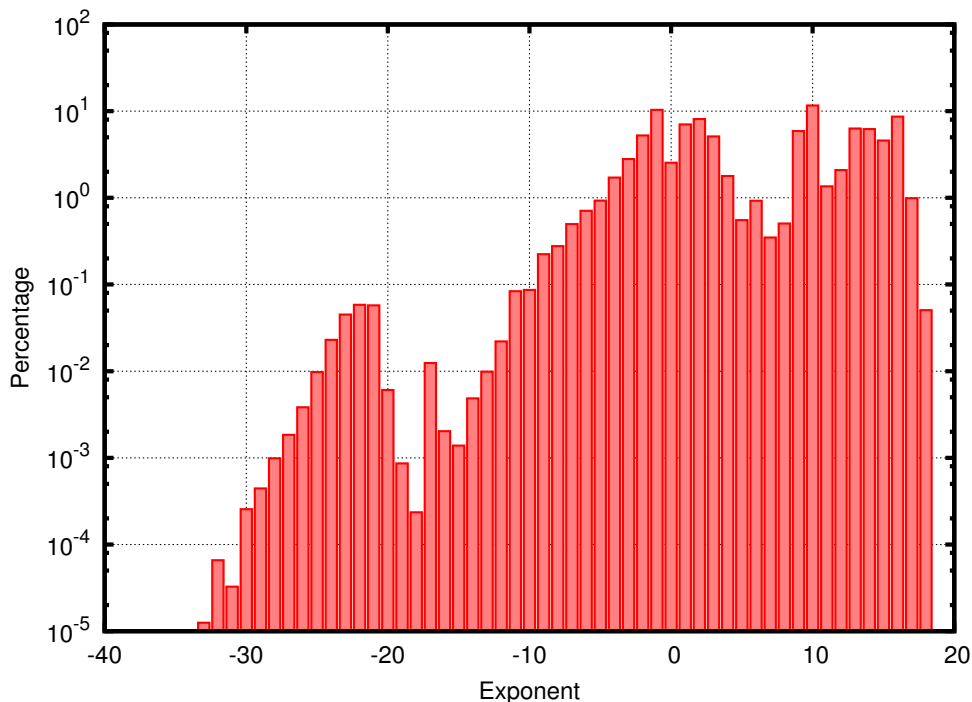


Figure 3.5. Distribution of values by floating point exponent. Taken from 183,561,915 floating point numbers representing the coordinates for the ray origins, ray directions, and triangle vertices. The smallest nonzero absolute value was 2.332×10^{-18} and the largest was 316,290.

or less, unless the scenes are first rescaled to an appropriate size.

Floating point numbers represent these values well, however, and using our reduced precision floating point library, we created a program to read in the ray-triangle pairs from the trace file and, using lower precision arithmetic, evaluate each of the $36 + 384 + 36 + 36 + 36 = 528$ scalar triple products that compute the V , V_a , V_0 , V_1 , and V_2 signed volumes, respectively. While each set of scalar triple products is geometrically equivalent, in practice, they may have wildly different characteristics due to catastrophic cancellation and other forms of numeric loss. One of our goals was to find the scalar triple product in each set that minimizes this. Note that we used the standard factored form for evaluating the scalar triple products rather than distributing the dot or cross products. Distributing leads not just to more operations to evaluate, thus making it slower, but also to more terms to sum which increases the potential for cancellation.

To evaluate the quality of these values, we also computed the five signed volumes using the rational arithmetic module of the GNU Multiple Precision Arithmetic Library [29]. Because the floating point values in the records can be losslessly converted to large rationals, and the computation of the signed volumes uses only addition, subtraction and multiplication which is exact with rationals, we can compute a perfect “gold standard” for what these values should be.

Losslessly converting the low precision values to rationals and taking the difference from the reference values allows us to accurately measure the error. For a given exponent and significand width, we measured the average relative error (relative error being the absolute value of the difference divided by the true value) for each scalar triple when computed with the reduced precision arithmetic. In the case of positive or negative INFs, we handled them by converting them to the largest or smallest representable numbers when calculating the relative error.

3.8 Significand and Exponent Widths

Tables 3.2, 3.3, and 3.4 show the results, with the average relative error for the best scalar triple products for computing the determinant, V , the volume for computing the ray parameter, V_a , and the volumes for computing the barycentric coordinates, $V_{0..2}$, respectively. Note that these last are considered together because cycling the triangle vertices has the effect of cycling the barycentric coordinates. The entries with eight bits of exponent and 23 bits of significand correspond to standard single-precision floating point arithmetic.

One important thing revealed by this analysis is that the width of the exponent has no effect on the error rate once the exponent field has sufficient range to avoid overflowing or underflowing during the computations. While single precision numbers use eight-bit exponents, seven bits appear to be perfectly sufficient for the scenes that we tested. Six bits or less is too few, however.

With sufficient exponent bits, the dominant factor in the error rate is the

Table 3.2. Relative error for determinant by significand and exponent widths. Each value shows the lowest average relative error for the volume V for a given significand and exponent width from among all the scalar triple products that compute it.

Significand	Exponent					
	6	7	8	9	10	11
8		0.0041931	0.0041931	0.0041931	0.0041931	0.0041931
9	0.0723670	0.0023152	0.0023152	0.0023152	0.0023152	0.0023152
10	0.0685687	0.0014908	0.0014908	0.0014908	0.0014908	0.0014908
11	0.0806531	0.0007918	0.0007918	0.0007918	0.0007918	0.0007918
12	0.0681592	0.0008845	0.0008845	0.0008845	0.0008845	0.0008845
13	0.0673424	0.0002177	0.0002177	0.0002177	0.0002177	0.0002177
14	0.0657871	0.0001257	0.0001257	0.0001257	0.0001257	0.0001257
15	0.0692937	0.0000522	0.0000522	0.0000522	0.0000522	0.0000522
16	0.0666937	0.0000290	0.0000290	0.0000290	0.0000290	0.0000290
17	0.0666923	0.0000173	0.0000173	0.0000173	0.0000173	0.0000173
18	0.0664176	0.0000085	0.0000085	0.0000085	0.0000085	0.0000085
19	0.0663998	0.0000039	0.0000039	0.0000039	0.0000039	0.0000039
20	0.0663904	0.0000020	0.0000020	0.0000020	0.0000020	0.0000020
21	0.0663843	0.0000012	0.0000012	0.0000012	0.0000012	
22	0.0663836	0.0000005	0.0000005	0.0000005		
23	0.0663804	0.0000003	0.0000003			
24	0.0663800	0.0000001				
25	0.0663797					

Table 3.3. Relative error for distance by significand and exponent widths. Each value shows the lowest average relative error for the volume V_a for a given significand and exponent width from among all the scalar triple products that compute it.

Significand	Exponent					
	6	7	8	9	10	11
8		0.0565310	0.0565309	0.0565309	0.0565309	0.0565309
9	0.2535980	0.0310891	0.0310889	0.0310889	0.0310889	0.0310889
10	0.2252270	0.0035108	0.0035108	0.0035108	0.0035108	0.0035108
11	0.3035140	0.0032066	0.0032066	0.0032066	0.0032066	0.0032066
12	0.4292930	0.0022818	0.0022818	0.0022818	0.0022818	0.0022818
13	0.9842460	0.0018787	0.0018787	0.0018787	0.0018787	0.0018787
14	1.1524700	0.0017955	0.0017955	0.0017955	0.0017955	0.0017955
15	1.8469000	0.0023425	0.0023425	0.0023425	0.0023425	0.0023425
16	3.9446000	0.0019844	0.0019844	0.0019844	0.0019844	0.0019844
17	6.2982300	0.0016163	0.0016163	0.0016163	0.0016163	0.0016163
18	8.9045900	0.0018556	0.0018556	0.0018556	0.0018556	0.0018556
19	16.6392000	0.0017347	0.0017347	0.0017347	0.0017347	0.0017347
20	41.9825000	0.0015805	0.0015805	0.0015805	0.0015805	0.0015805
21	80.0268000	0.0014150	0.0014150	0.0014150	0.0014150	
22	188.3990000	0.0013232	0.0013232	0.0013232		
23	395.6200000	0.0007361	0.0007361			
24	395.6200000	0.0004270				
25	395.6200000					

Table 3.4. Relative error for barycentrics by significand and exponent widths. Each value shows the lowest average relative error for the volumes V_0 , V_1 , and V_2 for a given significand and exponent width from among all the scalar triple products that compute it.

Significand	Exponent					
	6	7	8	9	10	11
8		0.0376633	0.0376633	0.0376633	0.0376633	0.0376633
9	0.0876891	0.0286147	0.0286147	0.0286147	0.0286147	0.0286147
10	0.0755282	0.0144344	0.0144344	0.0144344	0.0144344	0.0144344
11	0.0703083	0.0068975	0.0068975	0.0068975	0.0068975	0.0068975
12	0.0805000	0.0033597	0.0033597	0.0033597	0.0033597	0.0033597
13	0.0737002	0.0019055	0.0019055	0.0019055	0.0019055	0.0019055
14	0.0712016	0.0010664	0.0010664	0.0010664	0.0010664	0.0010664
15	0.0726447	0.0003917	0.0003917	0.0003917	0.0003917	0.0003917
16	0.0756132	0.0002467	0.0002467	0.0002467	0.0002467	0.0002467
17	0.0712252	0.0001681	0.0001681	0.0001681	0.0001681	0.0001681
18	0.0700354	0.0000577	0.0000577	0.0000577	0.0000577	0.0000577
19	0.0711816	0.0000355	0.0000355	0.0000355	0.0000355	0.0000355
20	0.0753862	0.0000141	0.0000141	0.0000141	0.0000141	0.0000141
21	0.0695343	0.0000098	0.0000098	0.0000098	0.0000098	
22	0.0704502	0.0000033	0.0000033	0.0000033		
23	0.0701544	0.0000035	0.0000035			
24	0.0701527	0.0000021				
25	0.0701514					

number of bits in the significand. Each additional bit added to the significand appears to roughly halve the error rate. The determinant, volume V , is the most robust value to compute with few bits, followed by the volumes $V_{0\dots 2}$ for the barycentric coordinates. The ray parameter (distance) calculation, V_a , is by far the most sensitive.

3.9 Robustly Computing Signed Volumes

For computing V , we found that the lowest average relative error was produced by the formula

$$V = [(\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_1 - \mathbf{p}_2)] \cdot \mathbf{d} \quad (3.3)$$

and the three equivalent formulas produced by reversing edges and/or commuting the cross product. Reversing edges commutes the operands to the subtractions which IEEE 754 guarantees affects only the sign of the outcome. Commuting the cross product commutes operands to multiplications, but IEEE 754 also guarantees that multiplication is commutative. Note that addition is not associative under IEEE 754; we evaluated all dot products by summing the products of the x and y components first, then adding the products of the z components. The worst formulas were anywhere from 6.7 to 25.8 times worse, depending on the significand size and all had the ray direction vector, \mathbf{d} inside the cross product. We would have expected variations of the best formula produced by cycling the triangle vertices to have produced similarly good results. Interestingly, there was considerable variation here, with the other two orders being 1.7 and 4.7 times worse. This may indicate a bias in how the models were tessellated into triangles.

For the ray parameter volume, V_a , we found the best results were produced by the formula:

$$V_a = [(\mathbf{p}_2 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_1)] \cdot (\mathbf{a} - \mathbf{p}_2) \quad (3.4)$$

and the seven equivalent formulas produced by reversing edges and/or commuting the cross product. The superiority of these expressions were not as

clear cut as for the determinant, but these appeared most often when examining the formulas with the lowest error rates. Amazingly, we found that the worst formulas were 17.0 times worse with 24 bits of significand, and were nearly 910.6 times worse with 11 bits of significand (and produced numeric overflows at ten bits or fewer when using a seven bit exponent.) One common feature of the bad formulas was that they used tetrahedral edges ending at the ray origin, \mathbf{a} , on both the inside and the outside of the dot product.

Finally, for computing the volumes $V_{0..2}$ determining the barycentric coordinates, the lowest error rates were most commonly produced by the formulas:

$$\begin{aligned} V_0 &= [(\mathbf{a} - \mathbf{p}_2) \times \mathbf{d}] \cdot (\mathbf{p}_1 - \mathbf{p}_2) \\ V_1 &= [(\mathbf{a} - \mathbf{p}_0) \times \mathbf{d}] \cdot (\mathbf{p}_2 - \mathbf{p}_0) \\ V_2 &= [(\mathbf{a} - \mathbf{p}_1) \times \mathbf{d}] \cdot (\mathbf{p}_0 - \mathbf{p}_1) \end{aligned} \quad (3.5)$$

and the three equivalent formulas for each one produced by reversing edges and/or commuting the cross product. The variant formulas:

$$\begin{aligned} V_0 &= [(\mathbf{a} - \mathbf{p}_1) \times \mathbf{d}] \cdot (\mathbf{p}_1 - \mathbf{p}_2) \\ V_1 &= [(\mathbf{a} - \mathbf{p}_2) \times \mathbf{d}] \cdot (\mathbf{p}_2 - \mathbf{p}_0) \\ V_2 &= [(\mathbf{a} - \mathbf{p}_0) \times \mathbf{d}] \cdot (\mathbf{p}_0 - \mathbf{p}_1) \end{aligned} \quad (3.6)$$

and their equivalents also produced the best results for certain significand widths, though not as commonly as the first set. Unlike computing the determinant, keeping the ray direction, \mathbf{d} , inside the cross product proves beneficial, so long as the ray origin, \mathbf{a} , is also there. Here the ratio in error rate between the worst and the best ranged from 66.7 to 332.2. Using nine bits or fewer in the significand result in overflows when using a seven-bit exponent. Similar to the computing the ray parameter, the worst formulas all had tetrahedral edges ending at the ray origin on both the inside and the outside of the dot product.

Another important aspect for the computation of the barycentric-related volumes is their signs, given that one method of performing the inside/outside test is examining the three barycentric coordinates for consistent signs. Out of

36,711,852 barycentric-related volumes computed, the formulas above erred on the sign only two and four times, respectively, when computed with a seven-bit exponent and 24-bit significand. The worst formula erred 237 times.

3.10 Designing a Robust Algorithm

Equations 3.5 and 3.6 show remarkable similarity. Each corresponding scalar triple product in them involves the ray direction, an edge of the triangle, and vector from one of the end points of that edge to the ray origin. The only difference is in which end point is used – both are valid, leaving freedom to choose which end point to use.

When two triangles share an edge, if we choose the same end point when evaluating the two barycentric coordinate volumes associated with that edge then except for the sign, we will always get exactly the same value for a given ray. Essentially, both triangles will perform identical arithmetic for the shared edge. Even if there is numerical loss, the tests for both triangles will agree on the value.

We can exploit this by defining an ordering for the triangle vertices. Define $\min(\mathbf{l}, \mathbf{r})$ to return the lexicographically smaller of its two operands:

$$\min(\mathbf{l}, \mathbf{r}) = \begin{cases} \mathbf{l} & \text{if } l_x < r_x \\ \mathbf{r} & \text{else if } l_x > r_x \\ \mathbf{l} & \text{else if } l_y < r_y \\ \mathbf{r} & \text{else if } l_y > r_y \\ \mathbf{l} & \text{else if } l_z < r_z \\ \mathbf{r} & \text{else if } l_z > r_z \\ \mathbf{l} & \text{otherwise} \end{cases}$$

and use this to choose between Equations 3.5 and 3.6 for computing each of the barycentric coordinate volumes. Compute volumes V_0 , V_1 , and V_2 this way and then compare the signs. If all three have the same sign then ray passes inside the triangle and the determinant and ray parameter can be computed with Equations 3.3 and 3.4. Figure 3.6 shows the complete algorithm in pseudocode.

```

function RayTriangle(a, d, p0, p1, p2)
  V0 ← [(a - min(p1, p2)) × d] · (p1 - p2)
  V1 ← [(a - min(p2, p0)) × d] · (p2 - p0)
  V2 ← [(a - min(p0, p1)) × d] · (p0 - p1)
  if (V0 < 0 or V1 < 0 or V2 < 0) and (V0 > 0 or V1 > 0 or V2 > 0) then
    return miss
  N ← (p2 - p0) × (p1 - p2)
  V ← N · d
  if V0 = 0 and p1 = min(p1, p2) or           ▷ Optional test for unusual cases
    V1 = 0 and p2 = min(p2, p0) or
    V2 = 0 and p0 = min(p0, p1) or
    |V| < ε then
      return miss
  Va ← N · (p2 - a)
  t ← Va / V
  β ← V1 / V
  γ ← V2 / V
  return hit, t, β, γ

```

Figure 3.6. Pseudocode for robust ray-triangle intersection algorithm.

By using the signs of $V_{0..2}$ for the inside/outside test and computing them consistently across shared triangle edges, the algorithm ensures that if a ray passes to the side of a shared edge, the tests for the two triangles will always see it as passing on the same side. The ray can intersect one triangle, but not both. Nor can it fall between them, unless a volume is exactly zero.

The optional test noted in the pseudocode handles this subtle case by giving priority to one of the triangles based on the winding of the triangle vertices. These cases are so rare however, as to not be worth the special handling.

The test for the case $|V| < \epsilon$ when the ray hits the triangle nearly edge-on is also unnecessary in practice. If t is positive, it will grow quite large in this case (possibly to INF), and the intersection will tend to be discarded as being behind other primitives or beyond the end of the valid ray parameter interval.

The key benefit to this algorithm is that it emphasizes correct handling of shared edges between triangles. The vast majority of edges are shared this way in triangle mesh models, making this a desirable property. Furthermore, by

using the inside/outside test to ensure that a ray can hit only one of a pair of adjacent triangles, it downplays the role of the ray parameter in deciding which of the two triangles has the closest intersection when the ray passes near the common edge. Given that Table 3.3 shows that the ray parameter is by far the least accurate value from the ray-triangle intersection, this is a useful property.

In terms of arithmetic operation counts versus the Möller-Trumbore algorithm, we estimate that an efficient implementation of this robust algorithm is 35 to 42% more expensive to compute. This may be worth the trade off, however, for batch renderers that wish to avoid artifacts and in custom hardware systems that use this to compensate for smaller, faster floating point functional units.

Tables 3.5 and 3.6 show how this algorithm performs on rendering the Stanford Bunny and Sponza Atrium models for different significand widths from 8 to 22 bits, and a seven bit exponent. We produced these images by modifying a small ray tracer to use our reduced precision library throughout in every place where standard floating point types had been used. Though the Möller-Trumbore algorithm is considered fairly stable numerically, our new robust ray-triangle intersection algorithm matches or exceeds it in every case.

Note that the reduced precision library was also used in the the construction of a median-split BVH and in traversal with the Williams et al. ray/box intersection algorithm [104] for rendering these images. Part of the reason for this was wanting to see how far we could reduce the precision of the floating point arithmetic throughout a ray tracer while still getting acceptable results. Single precision IEEE 754 floats, at four bytes with 23 bits of significand and eight bits of exponent appear to be more than adequate for these scenes. “Half-precision” two byte values with ten bits of significand and five bits of exponent are clearly too small. Our results, however, suggest that a three byte floating point format with a 16-bit significand and seven bit exponent may be a suitable compromise for low precision interactive ray tracing.

Table 3.5. Comparison on Stanford Bunny with various significant widths. Pixels more than 10% different in color from the corresponding pixels when rendered with standard double precision arithmetic count as errors.


































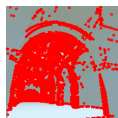














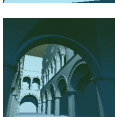
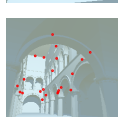
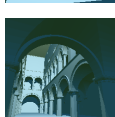

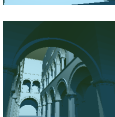

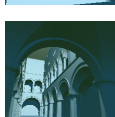

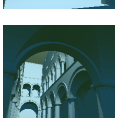
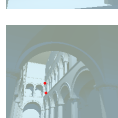
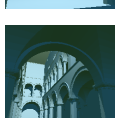

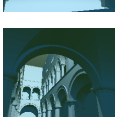
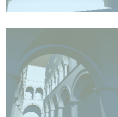
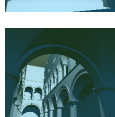

Width	Möller-Trumbore			Robust Algorithm				
	Render	Errors	PSNR	Render	Errors	PSNR		
8			3083	20.1			1464	23.6
10			590	27.5			161	32.5
12			186	32.8			45	37.5
14			37	40.9			11	42.7
16			13	47.5			3	53.1
18			1	62.2			0	75.3
20			0	82.5			0	81.0
22			0	92.1			0	92.1

Table 3.6. Comparison on Sponza Atrium with various significant widths. Pixels more than 10% different in color from the corresponding pixels when rendered with standard double precision arithmetic count as errors.

Width	Möller-Trumbore			Robust Algorithm				
	Render	Errors	PSNR	Render	Errors	PSNR		
8			6248	17.1			5259	17.8
10			3681	19.3			2197	22.2
12			391	30.9			197	34.8
14			135	35.0			98	36.5
16			22	44.0			11	46.5
18			10	43.8			7	46.6
20			2	60.8			1	62.6
22			0	92.1			0	92.1

3.11 Conclusion

We have presented two methods for optimizing ray triangle intersection for speed. Both of these differ from most previous approaches in that they are targeted toward implementations with ray packets. The first is based on simple operation counts. The second uses a more empirical approach and is probably more practical given the complexities of modern processors and compilers. In addition, the second uses knowledge from the first to improve performance further. An interesting question is whether the genetic algorithm approach can be extended to other components of ray tracing programs. Another question is whether the direct 3D approach examined here is not as efficient as the hit plane and 2D approach.

We have also explored the issue of numerical robustness with respect to the direct 3D approach and used it to design an algorithm that is more robust against errors introduced by low precision arithmetic. We expect that this algorithm will have two applications: first, in production batch renderers that can not afford to have any artifacts, and second, in custom ray tracing hardware with low precision floating circuitry.

CHAPTER 4

BETTER GRADIENT NOISE

This work is based on an earlier work: Hardware-Accelerated Gradient Noise for Graphics, in Proceedings of the 19th ACM Great Lakes symposium on VLSI, © ACM, 2009. <http://doi.acm.org/10.1145/1531542.1531647>.

A synthetic noise function is a key component of most computer graphics rendering systems. This pseudo-random noise function is used to create a wide variety of natural looking textures that are applied to objects in the scene. To be useful, the generated noise should be repeatable while exhibiting no discernible periodicity, anisotropy, or aliasing. However, noise with these qualities is computationally expensive and results in a significant fraction of the run time for scenes with rich visual complexity.

We propose three modifications to the standard algorithm for computing synthetic noise that improve the visual quality of the noise. First, a small change in the permutation hash function combined with separate pseudorandom tables yields significantly better axial decorrelation. Second, a modification to the reconstruction kernel approximating a global higher-order differencing operator produces better bandlimitation. Lastly, the quality of 2D surfaces using solid 3D noise is improved by reconstructing the stencil as projected onto the plane of the sample point and surface normal. These three techniques are mutually orthogonal, generalize to higher dimensions, and are applicable to nearly any gradient noise, including simplex noise. Combining them yields a desirable Fourier spectrum for graphics applications.

Next, we present a parallel hardware implementation of this improved noise function that allows the use of reduced precision arithmetic during the noise

computation. The result is a special-purpose function unit for producing synthetic noise that computes high-quality noise values approximately two orders of magnitude faster than software techniques. The circuit, using a commercial CMOS cell library in a 65nm process, would run at 1GHz and consume $325\mu m \times 325\mu m$ of chip area.

The noise hardware proposed in this chapter has been explored as part of a special-purpose hardware architecture called TRaX [91], a multithreaded many-core processor designed for ray tracing [88, 103]. In this architecture many thread processors share larger special-purpose functional units (such as *inv-sqrt*, *FP-mult*, and *noise*) to increase performance and to amortize hardware costs. That architecture is specifically targeted at ray tracing, but our noise hardware could also be used in any graphics system where high-quality noise is used for shading calculations. Including noise hardware on an existing commodity graphics chip (GPU), could greatly increase performance for procedural texturing on those systems.

4.1 Introduction

Procedural methods have many advantages in computer graphics. By tweaking only a handful of parameters, a digital artist can quickly populate a scene with massive amounts of rich detail. Each object or texture generated this way may have a unique appearance without any obvious repetition (e.g., tiling a hand-drawn texture.) Moreover, procedural techniques trade computation for memory. This is important since as process technology scales, compute resources will increasingly outstrip memory speeds. For texturing surfaces, the memory reduction can be two-fold: first there is the simple reduction in texture memory itself. Second, 3D or “solid” procedural textures can eliminate the need for explicit texture coordinates to be stored with the models. However, in order to avoid uniformity and produce visual richness, a simple, repeatable, pseudo-random function is required. Noise functions meet this need.

Simply described, a noise function in computer graphics is an $\mathbb{R}^N \rightarrow \mathbb{R}$

mapping used to introduce irregularity into an otherwise regular pattern. With the introduction of his noise function, Perlin [75, 80] enumerated several ideal qualities for such a function. Ideally, a noise function should have (1) a narrow bandpass limit in the texture space, and (2) a statistical character that is both stationary (translation invariant) and isotropic (rotation invariant). Peachey [74], in his excellent overview of noise, added: (3) be a repeatable pseudo-random function for a given input, (4) have a known range of outputs, and (5) avoid exhibiting obvious periodicity.

Noise has been used to simulate an incredible variety of appearances. Published examples of noise-based procedural shaders include cumulus clouds, hurricanes, clouds with coriolis effects, fire, water ripples, wavy water, sedimentary rock, and moons with rayed craters [66], marble, oak wood, brick walls, ceramic tiles, volumetric smoke, and lens flares [4]. (Figure 4.1 shows a simple example demonstrating some of these.) Higher dimensional noise allows for time-varying animations. Noise has even been used to compute velocity fields to emulate the appearance of turbulent fluid flow [12]. High-end movie graphics also makes extensive use of noise: rendered effects for “The Perfect Storm” were said to have averaged approximately 200 noise evaluations per shading sample [76]. Noise is ubiquitous in movie imagery and as a result, Ken Perlin was awarded a Technical Academy Award for Perlin Noise in 1997.

4.2 Noise in Graphics

Peachey [74] provides a good survey of noise functions. Many of these can be seen ideally as filtered white noise, produced from the convolution of a kernel function with a sampled random noise process where the strength of each impulse is uncorrelated. In each case, the choice of the kernel function and sample distribution may vary.

One of the simplest possible noise functions for computer graphics is value noise. Conceptually, this is produced by randomly sampling a white noise function and then using a reconstruction filter kernel to interpolate between



Figure 4.1. Rendered scene exhibiting noise-based procedural textures. Perlin noise was used to generate the wood grain pattern, the marble pattern, and the irregularities in the bricks and to control the density of the volumetric smoke. 1.3 billion noise evaluations were computed to render this image, averaging 552 per shading sample. The renderer spent 37.2% of its execution time evaluating noise.

the samples. Lewis's sparse convolution noise [56] is one such example.

For kernels, common choices include separable tent filters (for bilinear or trilinear interpolation), radial Catmull-Rom splines, and radial Gaussians. In each case, so long as the samples of the random noise process are properly uncorrelated, the shape of the noise function's spectrum will be dominated by the kernel.

For efficiency, most implementations use samples taken along a regular lattice and a simple interpolating reconstruction filter. Each lattice vertex within the range of the reconstruction filter's support around the input point is mapped to a sample value by hashing its coordinates, and then these values are interpolated at the input point to compute the noise function's value. A good hash function can provide a very large volume of noise without obvious periodicity, while reducing memory capacity requirements. Value noise is simple to understand and can be efficient for low-order interpolants. However, it tends to suffer from a blocky, anisotropic appearance (Figure 4.2a), even with a more expensive higher-order interpolant.

To overcome this, Perlin [75, 80] introduced gradient noise. Though not immediately apparent from the usual formulation, gradient noises such as Perlin noise can also be understood in terms of this convolution. Instead of using the product of the reconstruction filter with a random scalar value at each sample point, the product of the filter with a randomly oriented linear gradient is employed. The lattice coordinates are hashed to a unit vector, and the dot product of this vector with the vector from the lattice point to the input point is used to multiply the value from the filter. Because the dot product is linear, we can partition the noise into a sum of convolutions, one per dimension. Each filter becomes the product of a smoothing function with the gradient along an axis. For 2D Perlin noise, this gives $h_x * G_x + h_y * G_y$ where G_x and G_y are jointly distributed and where the kernels are defined as $h_x = s(x)s(y)x$, $h_y = s(x)s(y)y$. Original Perlin noise [75] applies a clamped cubic Hermite curve for each $s(t)$; Perlin's improved noise [80] employs a fifth-order polynomial.

This effectively creates a set of overlapping, randomly oriented dipole functions (“surflets” in Perlin’s terminology). Though the vector operations increase the computational complexity, the gradients eliminate much of the blockiness and a narrower filter over fewer samples can be used. (Figure 4.2b) Nonetheless, it still exhibits noticeable anisotropy.

In 2001, Perlin [77] noted the advantage of understanding noise as convolution with a clean separation between signal and reconstruction in allowing us to better understand its behavior, and more easily formulate variations. To this

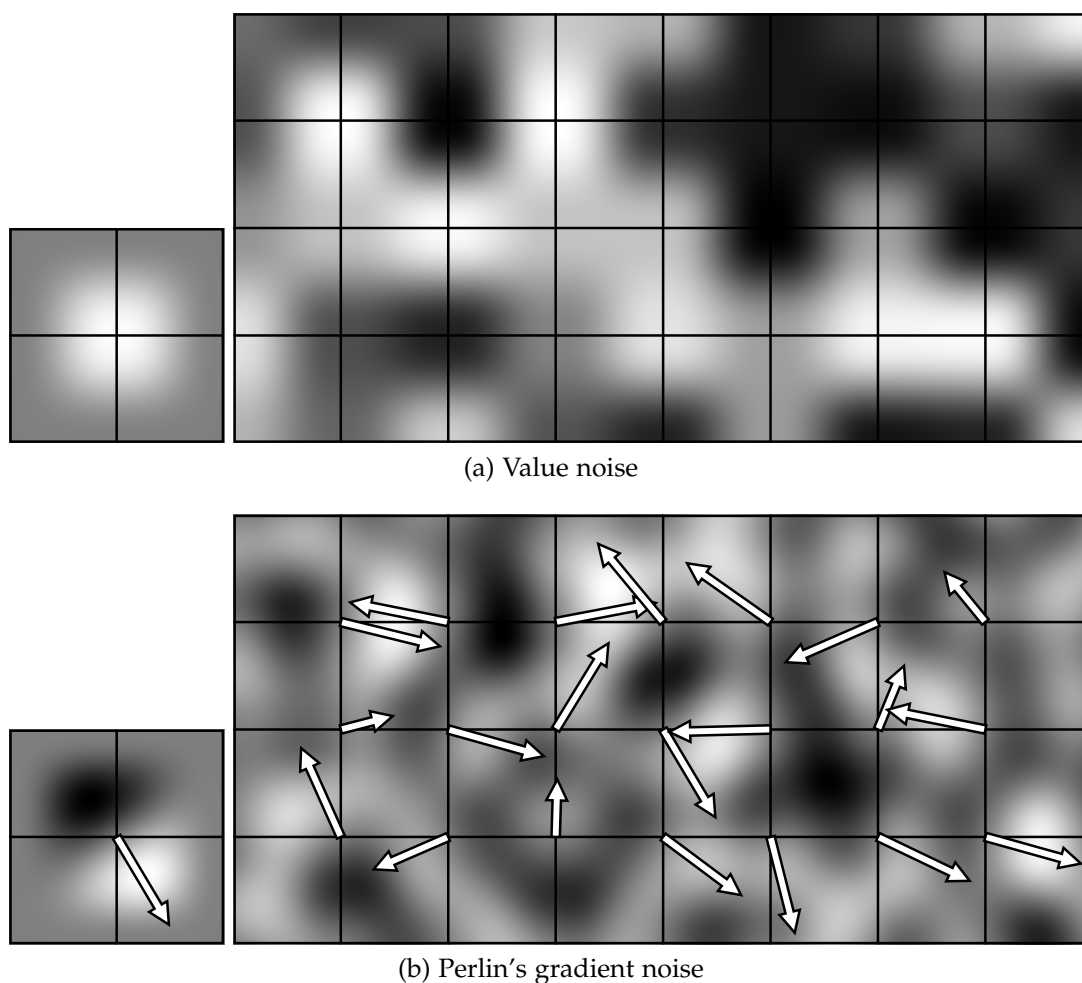


Figure 4.2. Value noise and Perlin’s gradient noise. To the left is the the reconstruction filter and an example of an oriented dipole, respectively. Gray represents zero, white indicates positive, and black is used for negative values. The grid overlay shows unit lengths in texture space. Arrows indicate gradient directions.

end, he introduced a more hardware-amenable variant of gradient noise known as simplex noise. Traditional implementations of Perlin noise had interleaved the sampling and reconstruction steps via interpolation. With simplex noise, Perlin made the noise more isotropic by substituting a radial reconstruction filter, $h = s(\sqrt{x^2 + y^2})$, for the previous separable one. The second change was to switch from a cubic lattice to a simplex lattice, thereby reducing the number of sample points evaluated during the reconstruction. The simplicial grid is also far more efficient for higher-dimensional noise. Finally, he introduced the idea of only using -1, 0 and 1 as components of the gradients in order to eliminate the multiplications in the dot products. However, this reduction in samples and simplification of the gradient calculations gives simplex noise a noticeably different visual quality.

In 2002, Perlin [78] returned to a more traditional noise function (with cubic grid and separable filter) with small adjustments to the interpolant and a clarification of simplified gradients from simplex noise. First, he switched to a higher-order polynomial for the interpolant in order to improve the appearance when Perlin noise is used for displacement mapping. Second, by changing from a table of random unit vectors to a set of vectors based on the midpoints of the edges of a cube, he further reduced the effective number of gradients to twelve. The regular distribution also eliminates the problem of clumping. An alternate solution would have been to apply a relaxation algorithm to the randomly generated unit vectors as a preprocess [79]. While improvements, these changes do not remove the axial correlation responsible for most of the anisotropy, and neither significantly attenuates lower frequencies (Figure 4.3a).

Cook and DeRose [19] noted that Perlin's noise still had several flaws and introduced an alternative to gradient noise, wavelet noise, to overcome these issues. The essence of their algorithm is to initially create an image of random noise, down-sample to half size, up-sample back to full size, and then subtract this result from the original. To evaluate the noise at a point, they filter the image with a uniform quadratic B-spline, in a process similar to evaluating value

noise. The initial construction of wavelet noise produces tighter band limits in its frequency distribution, both at the lower and at the upper limits. The result is orthogonal bands that allow for better spectral control. They also note the Fourier slice theorem is responsible for low frequencies “leaking” in when evaluating 2D slices embedded in a higher 3D noise volume. They solve this problem with a modification to the filtering step based on projection along the surface normal. Wavelet noise [19] succeeds in avoiding the axially-correlated periodicity of the traditional hash function, and ensures a desirable bandlimited spectrum. In practice, however, it is more costly than improved Perlin noise, and exhibits anisotropy in the lower bandlimit which may affect visual appearance when viewed closely (Figure 4.3b). To be efficient, wavelet noise also requires significantly more memory to store the complete preprocessed noise volume, whereas Perlin noise relies on a simple hashing scheme to generate the volume.

In the Sections 4.4 through 4.7, we propose several orthogonal improvements to standard gradient Perlin noise that combine the best features of each of these noise functions. We first improve both the generation of precomputed gradients and the spatial hash function to decrease axial bias. We then suggest a wider filter that greatly improves bandlimits as in wavelet noise, while preserving the visual characteristics of classic Perlin noise. Finally, we show how to take 2D projections of 3D Perlin noise to achieve desirable visual properties (Figure 4.3c).

4.3 Noise in Hardware

There has been some work published on hardware noise implementations [38, 69, 79]. This work does not actually propose special hardware for computing noise, instead it describes details to implement Perlin style noise using GPUs by mapping the lookup tables to texture memory. They are software adaptations of a noise algorithm to run on GPU hardware. While this approach is useful, it is quite different from our approach to hardware noise.

Our noise implementation is an actual parallel hardware implementation of noise as a custom circuit for use as a co-processor or as a functional unit to

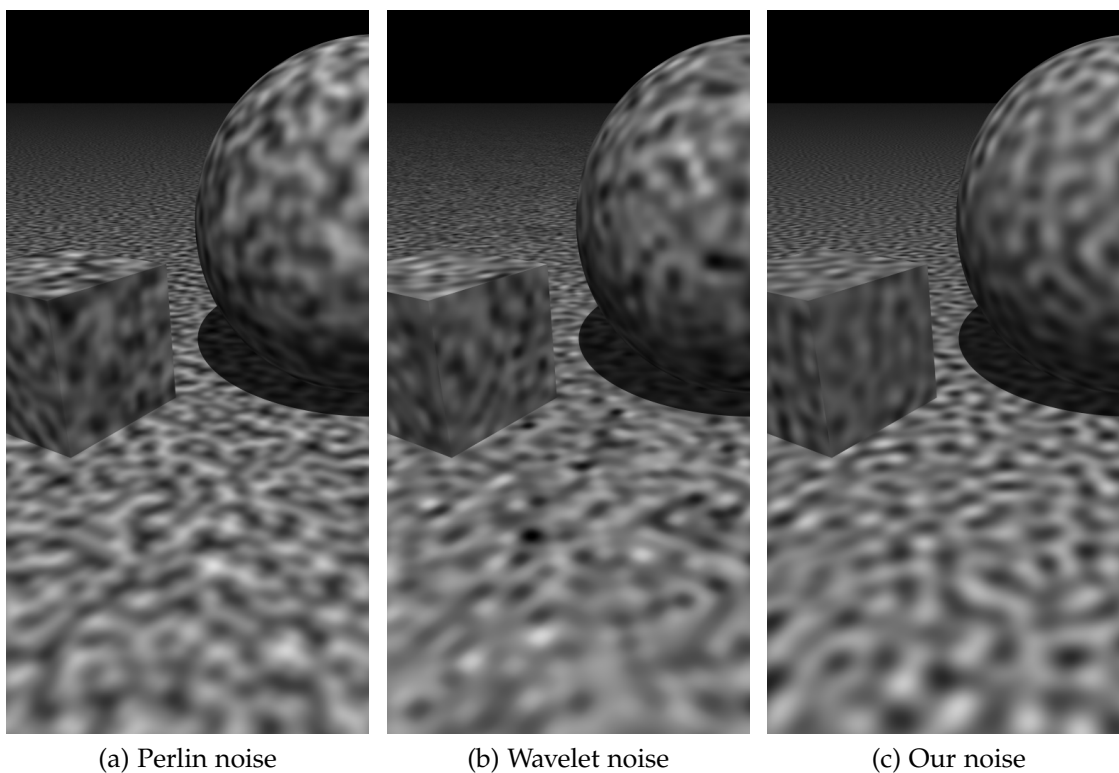


Figure 4.3. Simple scene comparing three noise algorithms. Sphere, box, and plane textured with (a) Perlin's improved noise function, (b) wavelet noise with projection to the surface normal, and (c) our modified noise function with improved gradient table, hash function, filter kernel and projection method. Our noise has a smoother look, appearing more isotropic close up and more even from mid to far.

be included in future designs. While this approach does not leverage existing high performance architectures like GPUs, it does have the potential to be used in GPUs to further increase the performance and quality of these kinds of computations.

As described in Section 4.2 Perlin designed his simplex noise to be more amenable to hardware acceleration by using a set of twelve fixed vectors with unit components to reduce the computation needed for the dot products. While this does reduce the required computation somewhat, it also creates some additional artifacts in the image that are quite apparent and which we wanted to avoid in our noise algorithm.

4.4 Gradient Vectors

Perlin's original noise function [80] used a table, \mathbf{G} , of random gradient vectors that were uniformly distributed on the unit sphere. These were constructed through the common rejection method of repeatedly picking random values from $[-1, 1]$ for each of the x, y, z components until the chosen vector had unit length or less. Once the table was filled, the vectors were normalized to unit length.

Later, Perlin observed that this tended towards clumping due to directional biases from the cubic grid [78]. To correct this, he introduced a fixed set of 12 gradient vectors formed from connecting the center of a cube to the center of each edge. For efficiency, four vectors that form a tetrahedron are duplicated to pad the table to 16 entries.

These new vectors lead to other artifacts, however, due to the symmetries between them. Axis-aligned planar cross sections of the 3D noise function will use only eight unique gradients due to four pairs of gradient vectors projecting to the same vector on the plane. Coupled with the duplicate vectors for padding, this can produce strong biases in a limited choice of gradient directions, as shown in Figure 4.4a. This leads to artifacts at 45° angles that highlight the underlying grid.

Instead, we use a larger table with Perlin’s relaxation idea [79] to solve the clumping problem. Beginning with the original gradient vector generation method, we then treat each of the vectors as a charged particle on the surface of a sphere [25]. Each particle representing a gradient vector, $\mathbf{G}[i]$, experiences a repulsive Coulumb force,

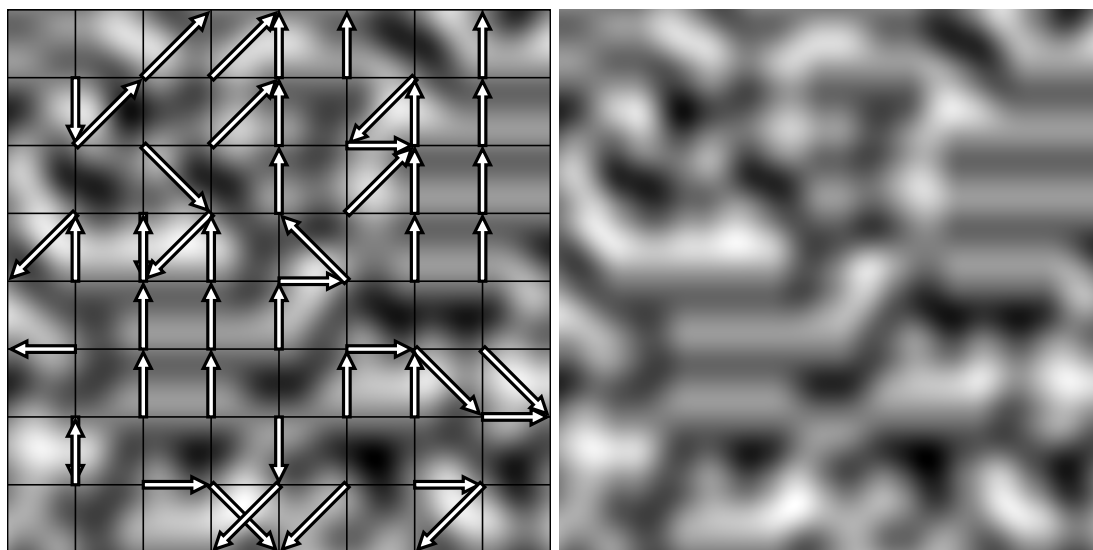
$$\mathbf{F} = \frac{\mathbf{G}[i] - \mathbf{G}[j]}{|\mathbf{G}[i] - \mathbf{G}[j]|^3},$$

from each of the other particles $j \neq i$. Simulating a series of time-steps to convergence that apply these forces to update the gradient vectors while confining them to the surface of the unit sphere produces a well-distributed table of gradient that avoids clumping and samples the surface of the unit sphere quite evenly. The improvements from the relaxation step can be quite subtle when the number of gradient vectors is large; nonetheless, as it can be done in a preprocess it is essentially free. Figure 4.4b shows noise with gradient vectors from a 256-entry table. Using a larger table avoids the quantization and symmetry effects of the cubic edge-center set and also provides an additional degree of randomization beyond that of the hash function.

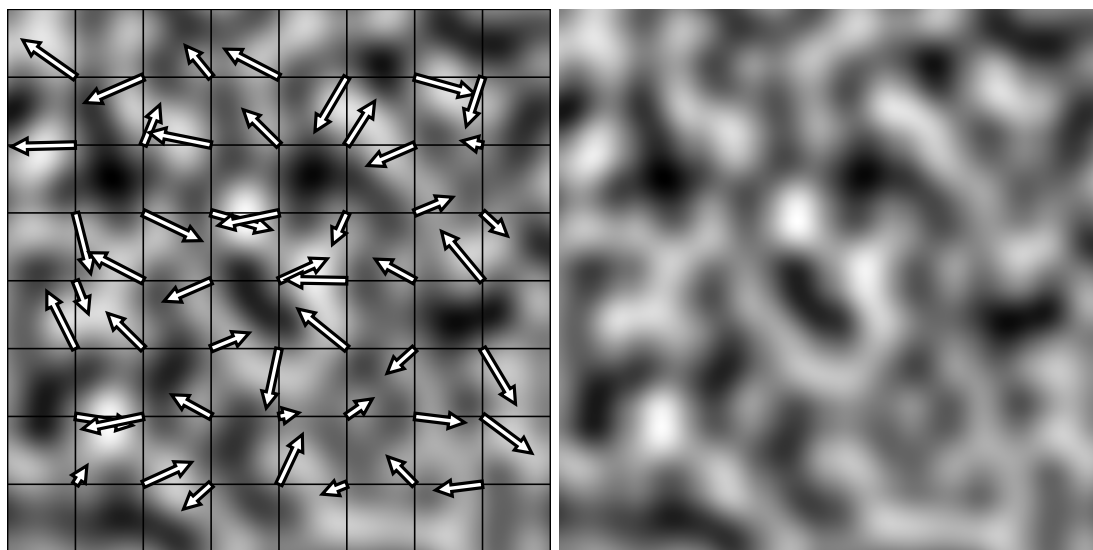
4.5 Hash Function

Perlin [80] generates random unit vectors on the 3D integer lattice indexed as i, j, k . To achieve finite storage, a single table P of N randomly permuted integers $0 \dots (N - 1)$ is hashed successively using each axial coordinate, $H_{ijk} = P[P[P[i] + j] + k]$, where indices are assumed to be modulo the table length. This provides the index into another table \mathbf{G} (modulo the size of \mathbf{G}) containing the gradient vectors.

The purpose of the hash is to decorrelate the indices. However, if i and j are held constant and steps are taken along k , this will unfortunately produce successive entries in P . For any values of $\{i, j, k\}$ that hash to $P[0]$, the adjacent lattice point $\{i, j, k + 1\}$ will always produce $P[1]$. In fact, each column will produce exactly the same sequence of hash values as any other – the copies will simply be shifted. This breaks the fundamental assumption that the samples of



(a) Cubic edge-center vectors



(b) Relaxed vectors

Figure 4.4. Projections of 3D gradient vectors onto planar cross section. Sections of axis-aligned 2D slices from a 3D noise volume with the grid and the gradient vectors projected onto the plane for (a) Perlin's set of 16 cubic edge-center vectors, and (b) 256 random vectors distributed via repulsion. Larger sets help avoid biases that can produce runs.

the random noise process are uncorrelated. Given a sufficiently large sampling of Perlin noise, the spectrum manifests strong striations perpendicular to the preferred axis, shown in Figure 4.5a.

To fix this, we use a separate (power of two sized) permutation table for each dimension, P_x, P_y, P_z , and take the exclusive-or of the value from each:

$$H_{ijk} = P_x[i] \oplus P_y[j] \oplus P_z[k].$$

Figure 4.5b shows the improvement to the spectrum that this produces. While requiring a slight increase in memory, this has the advantage of eliminating dependent permutation table lookups and reducing the total number of those lookups from 14 to 6 in the case of 3D Perlin noise. In the case of ND noise, the number of lookups is reduced from $2^{N+1} - 2$ to $2N$.

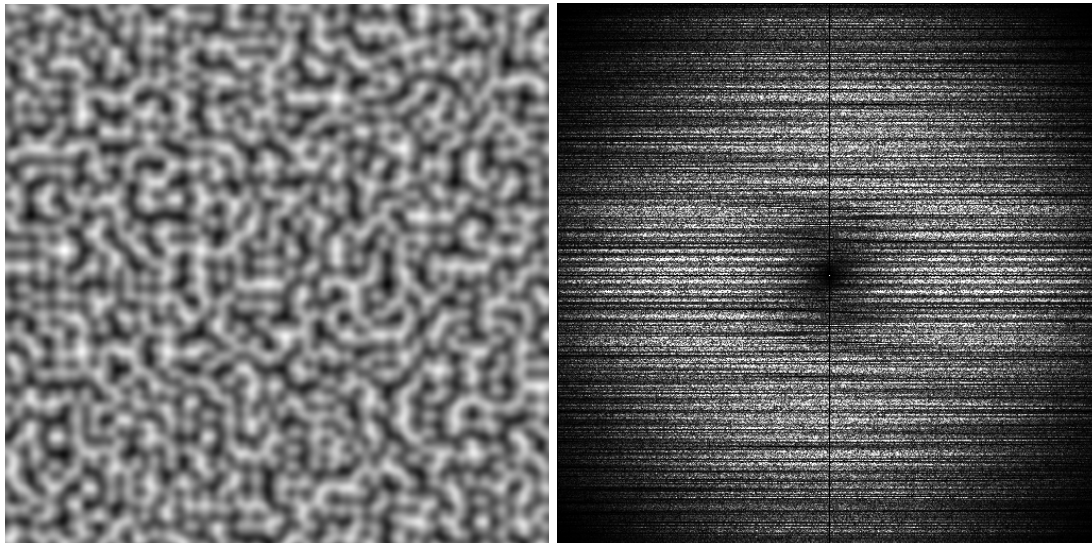
Note that if the set of gradient vectors is well-ordered, as in Perlin's newer 16-vector set, then an additional permutation table may be required to provide further scrambling of the index into \mathbf{G} . The random ordering of the vectors generated by the rejection and point repulsion methods provides this final scrambling inherently.

4.6 Filter Kernel

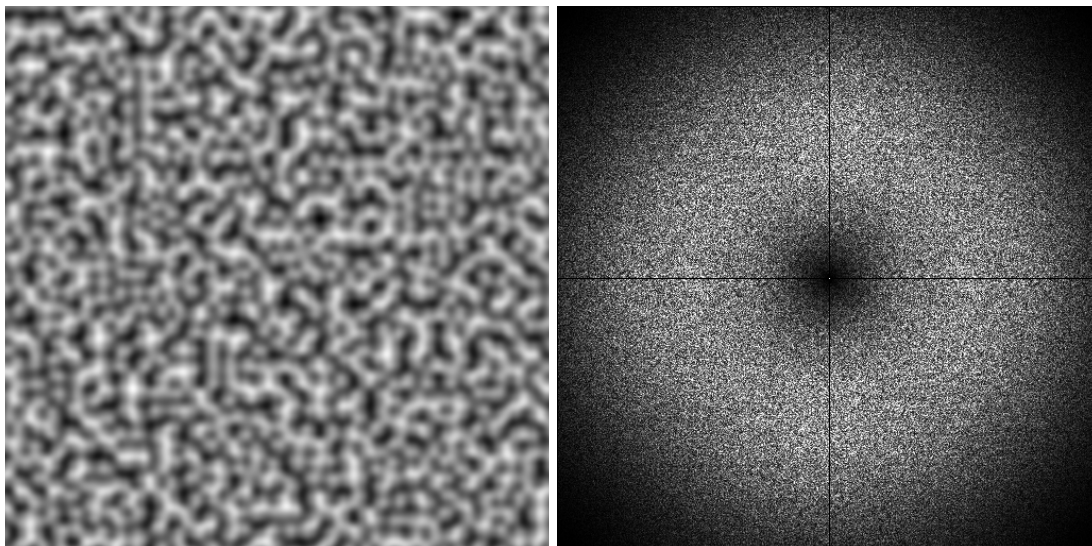
Though our hashing scheme spectrum fixes most of the anisotropy of the Fourier spectrum, the bandlimits are quite weak: both low and high frequencies remain. Provided that the random noise process is truly uncorrelated, the overall shape of the frequency spectrum is determined by the reconstruction kernel.

When considered in one dimension, Perlin noise uses an antisymmetric kernel of the form $s(x)x$ where $s(x)$ is either an Hermite cubic polynomial [80] or a quintic polynomial [78]. The latter produces a pair of lobes with opposite signs that peak at a distance of approximately 0.80 from each other and then fall to zero beyond that.

We note that the separable Perlin filter very closely resembles a continuous (smoothed) version of the discrete impulse response of a first-order (forward or backward) difference operator. Moreover, successive application of a difference



(a) Standard hash function



(b) *xor* hash function

Figure 4.5. Traditional and *xor* hash functions. Detail of images and associated Fourier transforms of (a) 2D Perlin noise with standard hash function, quintic interpolant and 256-entry gradient table, (b) 2D Perlin noise with our *xor* hash function. Our *xor* hash eliminates the striations in the Fourier transform.

operator k times to a given impulse causes an impulse response of the $k + 1$ set of binomial coefficients with alternating signs. Applied as a filter, this scales frequency f by $(2 \sin(\pi f))^k$ [40]. Effectively, differencing attenuates the lower third of the Nyquist interval and amplifies the upper two-thirds, explaining the strong presence of high frequencies and the near-linear attenuation of low frequencies in the Perlin noise spectrum. The smoothing component isolates the first replica and eliminates the higher order harmonics. Thus, higher order differencing with smoothing ensures better bandlimits, but at the cost of a wider filter.

Kernels with opposing lobes, such as a Sobel filter or Catmull-Rom spline, are common solutions to bias from discrete signal reconstruction. While Perlin's filter is shaped correctly, it lacks sufficient support width to bandlimit as desired. The lowest support k for which the stencil encloses the next immediately neighboring samples is $k = 3$, a four-point stencil on $[-2, 2]$. The binomial coefficients of third-order forward differencing are 1,-3,3,-1, so we desire additional opposing lobes one unit away and 1/3 the amplitude of the inner pair. For computational efficiency our filter approximates this as a polynomial $s(x)$; which is again applied to the antisymmetric Perlin noise kernel of the form $s(x)x$. In choosing $s(x)$ we seek a symmetric (even-degree) polynomial that makes $s(x)x$ satisfy the previous constraint on the lobes, and has $s' = s'' = 0$ at the stencil endpoints. We find that the following meets these criteria:

$$\begin{aligned} s(x) &= (2 - x)^4(2 + x)^4(1 - x)(1 + x)/256 \\ &= 4(1 - x^2/4)^5 - 3(1 - x^2/4)^4. \end{aligned}$$

We implement the function in the second form for efficiency. Figure 4.6 compares this with Perlin's quintic. Combining the radial filter kernel with our new hash function in 2D noises produces the results shown in Figure 4.7, in which high frequencies are effectively attenuated. The disadvantage of our method is that the four-point stencil is costly, requiring 16 lookups in 2D and 64 in 3D. However, this larger kernel allows contributions from the additional grid points

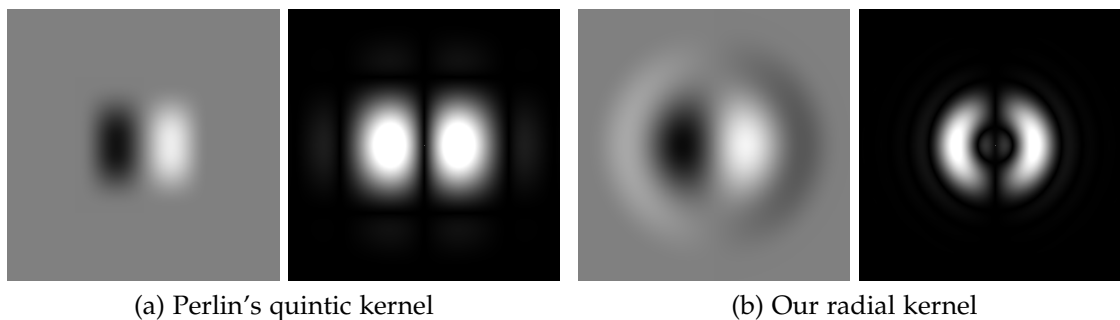


Figure 4.6. Separable and radial filter kernels. Images and associated Fourier transforms of (a) filter kernel from Perlin's improved quintic polynomial, and (b) our higher-order radial kernel with secondary lobes.

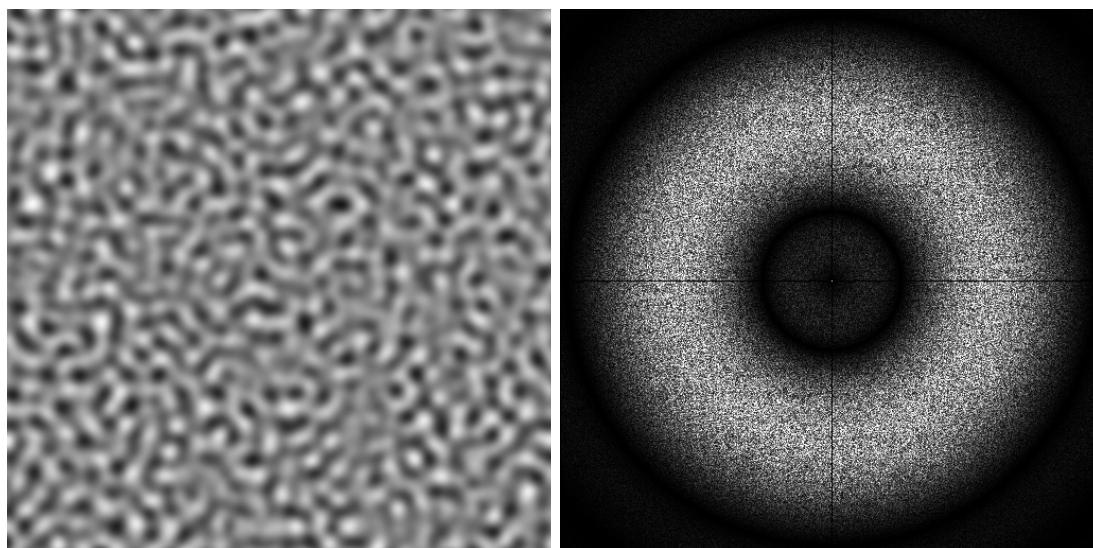


Figure 4.7. Gradient noise with extended reconstruction kernel. Detail of image and associated Fourier transform of 2D noise with 256 well-distributed gradient vectors and our *xor* hash function and extended reconstruction kernel.

to eliminate the regular zeroes of both classic Perlin noise and simplex noise, and is clearly effective in producing a band-pass spectrum.

4.7 Projection to 2D

In their use as solid textures, 3D noises are frequently sampled along 2D surfaces. However, Cook and DeRose [19] showed that even if a noise is 3D bandlimited, a planar slice will not be bandlimited due to the consequences

of the Fourier slice theorem. Instead, low frequencies will be present and the Fourier transform will appear as a solid disk (Figure 4.8). To solve this problem, they project the wavelet noise onto a surface by performing a weighted line integral along the surface normal under the assumption that the curvature is weak at the scale of the noise. Inspired by their method, we propose a simple projection technique applicable to gradient noises.

Our method projects each of the neighboring points onto the plane tangent to the surface, and evaluates the kernel at those projected points instead of at their original positions (Figure 4.9). Because of the radial nature of the kernel this results in a planar slice using a convolution kernel equivalent to the 2D kernel when evaluating the noise along a surface with low curvature. To compensate for the projection and avoid popping, we weight the contribution from each neighboring point with a cubic Hermite curve that falls off with the distance from the plane. Figure 4.10 shows our result: low frequencies are attenuated isotropically, yielding a more radially symmetric spectrum than wavelet noise.

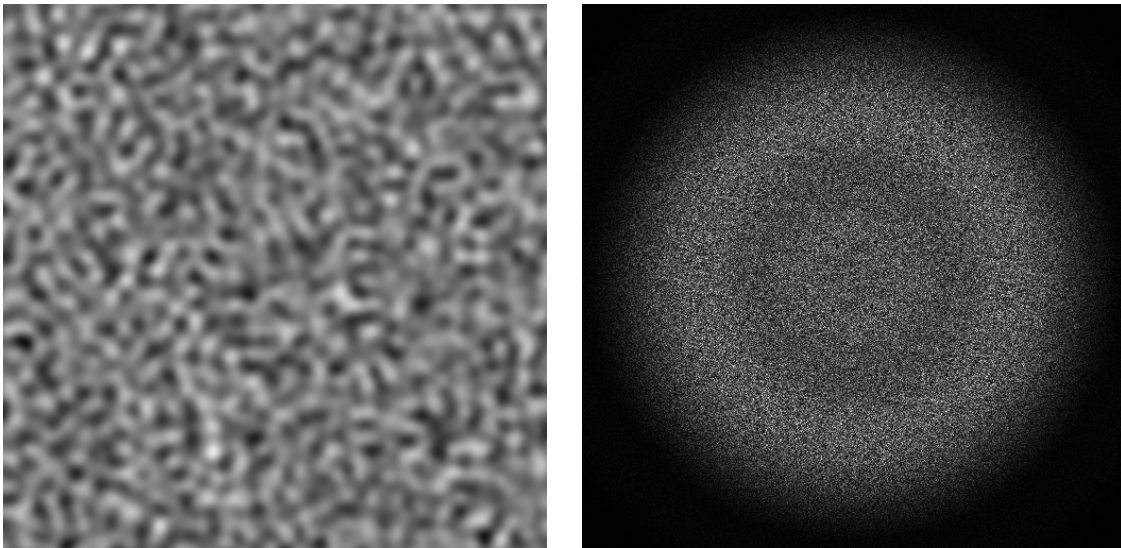


Figure 4.8. Low frequencies in planar slice. Detail of images and associated Fourier transforms of a slice of our 3D noise along plane normal to $x = y = z$.

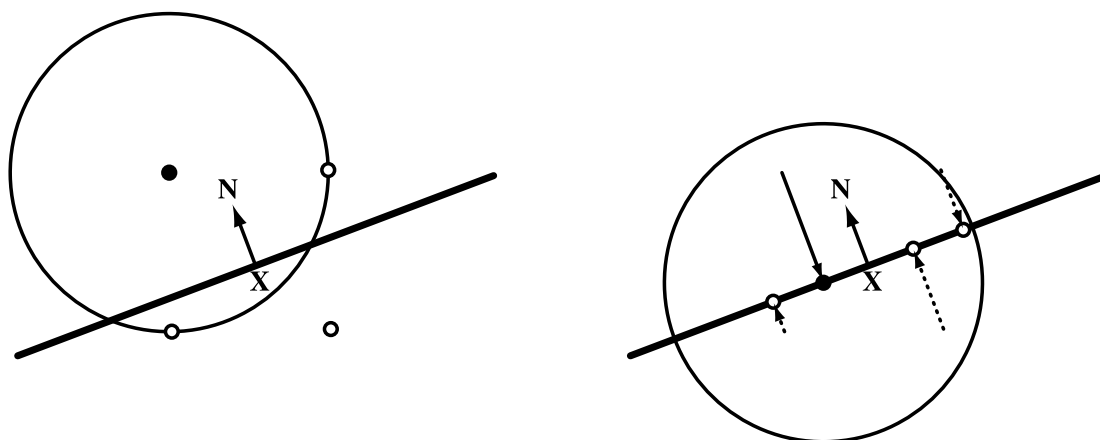


Figure 4.9. Projecting the lattice points. Without projection (left), cross sections of noise will produce off-center slices through the filter kernel. By projecting the lattice points onto the plane tangent to the surface normal (right) before evaluating the contribution, the slices pass through the kernel center.

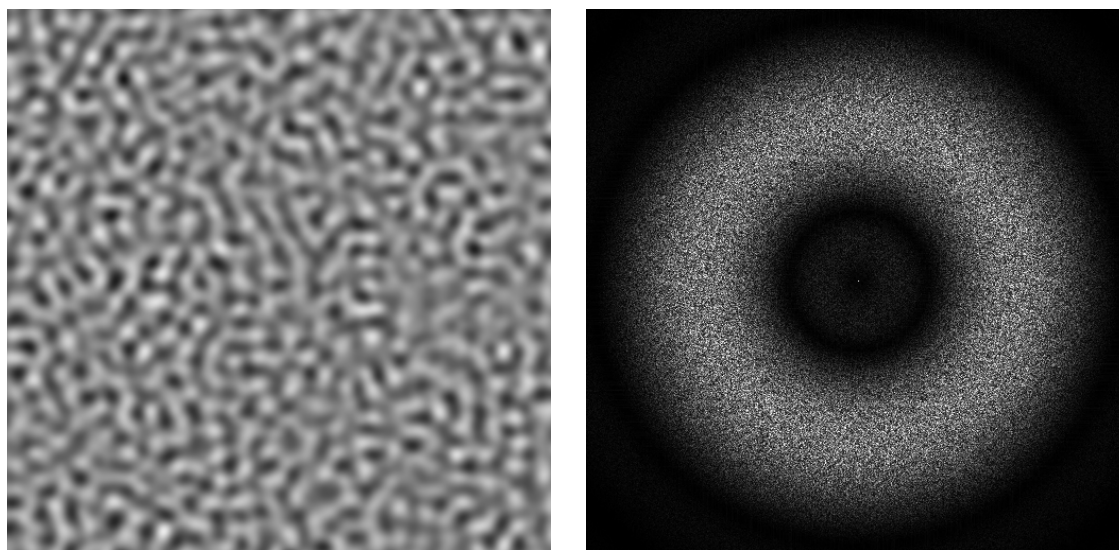


Figure 4.10. Low frequencies removed by projection. Detail of images and associated Fourier transforms of the same slice as in Figure 4.8 with the projection method applied.

4.8 Software Algorithm

Figure 4.11 displays complete pseudocode that employs our hash, filter, and projection method (given coordinate \mathbf{X} and unit-length surface normal \mathbf{N}). Indices into each table are assumed to be modulo table size.

We have shown how to improve the spectral properties of Perlin noise with relatively minor changes. Distributing the gradient vectors evenly over the sphere has no run-time or memory cost. The hash method of Section 4.5 improves both visual quality and runtime (about 5%) with only a modest increase in storage; we believe all noise implementations should adopt these changes.

The kernel and projection method yield a bandlimited spectrum attenuating high and low frequencies. Timings of 3D noise on a 2.5 GHz G5 indicate that these methods are more expensive to evaluate than standard Perlin noise (about $5.9\times$ and $13.8\times$, respectively), thus their use should be restricted to applications where the visual benefits merit the penalty. GPU implementations show consistent ratios.

```

function ProjectedNoise( $\mathbf{X}$ ,  $\mathbf{N}$ )
   $\mathbf{I} \leftarrow \{ \lfloor X_x \rfloor, \lfloor X_y \rfloor, \lfloor X_z \rfloor \}$ 
   $\mathbf{F} \leftarrow \mathbf{X} - \mathbf{I}$ 
   $v \leftarrow 0$ 
  for  $k \leftarrow -1$  to  $2$  do ▷ 4-point stencil
    for  $j \leftarrow -1$  to  $2$  do
      for  $i \leftarrow -1$  to  $2$  do
         $\mathbf{D} \leftarrow \mathbf{F} - \{i, j, k\}$  ▷ vector from lattice point
         $o \leftarrow \mathbf{D} \cdot \mathbf{N}$  ▷ offset from plane
         $\mathbf{A} \leftarrow \mathbf{D} - o\mathbf{N}$  ▷ projection onto plane
         $d \leftarrow \mathbf{A} \cdot \mathbf{A}$  ▷ squared distance to projection
         $o \leftarrow 1 - |o|$ 
        if  $d < 4$  and  $o > 0$  then
           $a \leftarrow 3o^2 - 2o^3$  ▷ attenuation for offset
           $h \leftarrow P_x[I_x + i] \oplus P_y[I_y + j] \oplus P_z[I_z + k]$ 
           $t \leftarrow 1 - d/4$ 
           $v \leftarrow v + (\mathbf{A} \cdot \mathbf{G}[h])(4t^5 - 3t^4)a$ 
  return  $v$ 

```

Figure 4.11. Pseudocode for the complete noise algorithm.

Our algorithm compares favorably against wavelet noise while providing many of its visual benefits. Tests against the provided sample code for Wavelet noise indicate that our modified noise is $3.4\times$ faster without projection and at least $4.5\times$ faster with it (significantly so in the nonaxis-aligned case). Our algorithm is also more parsimonious with memory: a 128^3 tile of noise requires 128MB of storage for wavelet noise and 4KB with our algorithm (with 256-entry gradient table and 128-entry permutation tables.) Furthermore, the memory needed for wavelet noise grows with the volume of the tile while the memory for ours grows linearly with the length of the sides.

While we have focused on visual quality, we note that with our method we can trade some visual appearance for speed and still retain the isotropic bandlimits by sampling the grid at half-resolution—i.e., using only the integer lattice points with even-valued coordinates (Figure 4.12). This variant is well suited to summations over multiple octaves and effectively returns to a two-point stencil and with projection it has an evaluation cost of only $2.4\times$ relative to Perlin noise. In either case, the fewer dependent lookups needed for the new hashing scheme implies opportunities for optimization.

4.9 Hardware Gradient Noise

We have used this modified gradient noise algorithm as the basis for our hardware noise implementation. We implemented the new hashing scheme, the new radial polynomial with the coarser grid evaluation, and generated the vectors in the gradient tables with the relaxation technique.

As a data point, one straightforward implementation of Perlin noise that we have measured requires 120 floating point operations to compute one noise value. Our higher quality noise requires as many as 172 operations in software, though many of these operations are independent of each other. Clearly if we can parallelize this process we can achieve a much faster noise implementation. An overview of the parallel hardware implementation of our improved noise algorithm can be seen in Figure 4.13. As mentioned above, our improved noise

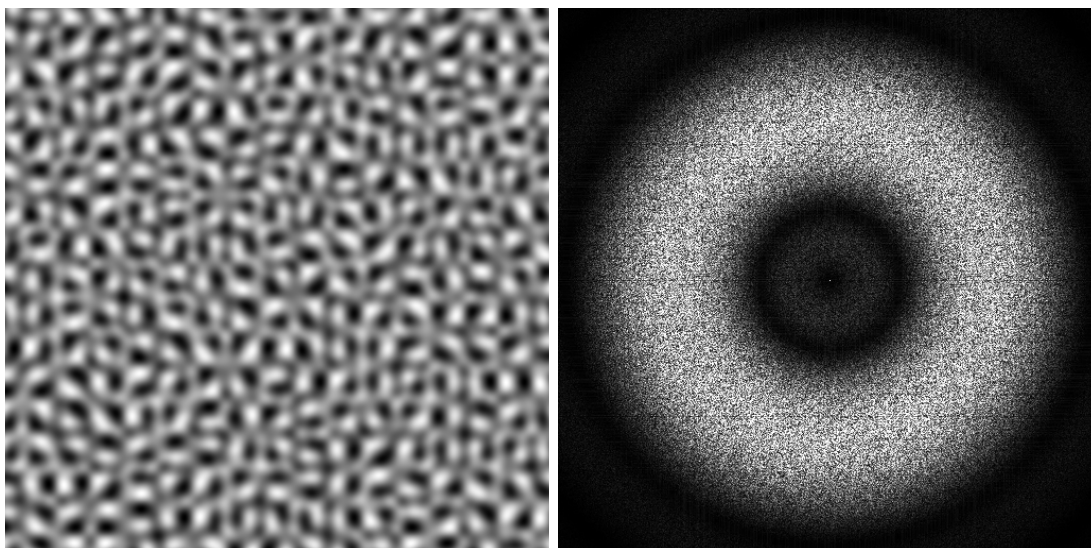


Figure 4.12. Noise evaluated on half-resolution grid. Detail of image and associated Fourier transform of same planar slice and projection method with faster half-resolution grid.

uses a gradient table to provide better results than either simplex noise or value noise. This improvement in quality comes at a cost of increased circuit size (to hold the table and to perform the full dot products). We believe it is a good trade off considering the improvement in quality. Additionally, we use reduced precision fixed point arithmetic to save area, energy and delay. We also achieved additional savings by reducing the sizes of the gradient and hash tables. As can be seen from Figures 4.1 and 4.14, our fixed point implementation, while distinguishable as being a different image, does not have a noticeable difference in quality from the standard floating point implementation used in software.

For our circuits we used standard cells from Artisan targeted to a 65nm CMOS process. We used Synopsys Design Compiler as a synthesis front end and Cadence SOC Encounter for back-end place and route. For table comparisons between standard cells and ROMs we used Artisan via-ROM generators for the same 65nm CMOS process.

The typical application for noise is generating images with an eight bit representation per color channel. This allows us to use much lower precision in

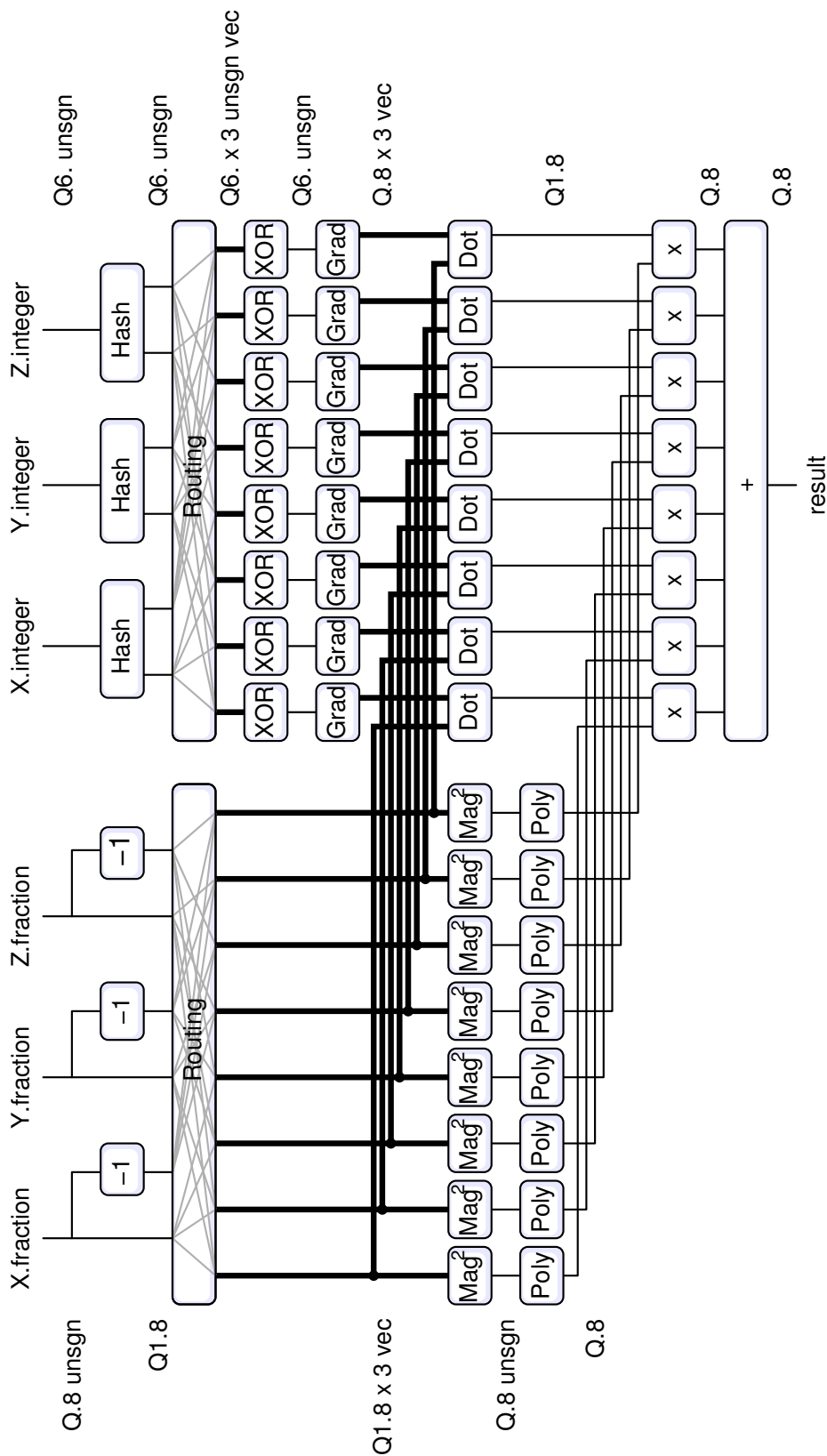


Figure 4.13. High-level diagram of our hardware noise implementation. Thick lines indicate vector values, thin lines are scalar values. All values are signed (two's complement form with an implicit additional high bit) unless noted as unsigned. The left side illustrates the computation of the polynomials on the squared magnitudes of the vector offsets. The right side shows hashing the integer lattice coordinates to lookup the gradient vectors for the dot products before combining these with the polynomials to produce the final value.



Figure 4.14. Rendered scene with fixed point version of our noise. Compare with Figure 4.1. All noise evaluations were performed with eight fractional bits and a 64-entry gradient table. While not identical, this shows that the modified algorithm with reduced precision can be a viable alternative to floating point Perlin noise.

our circuits than would be used in a traditional software implementation of noise. Even when the software version, or the GPU version, of noise does the full computation in 32 bit floating point most of the precision is thrown out in the end. We leverage this to use eight bit fixed point computations in our noise circuits which results in tremendous savings in area, power, and latency with no discernible visual artifacts. While multiplication operations do lose some precision due to truncation of the lower bits, the deepest chain of arithmetic operations contains just five multiplies.

4.10 Lookup Tables

There are a number of lookup tables used in our improved noise algorithm (permutation tables and gradient tables) and we explored a few options for implementing them. We first used the ROM generator, but the results required a fairly large area. We found that by implementing the lookups as case statements in Verilog and synthesizing to standard cells we were able to decrease the area needed for the lookup tables by a factor of 3.3. The latency through the ROM was also worse and forced us to latch the value at the output instead of allowing us to perform register retiming through the lookup tables. We believe this is because the lookup tables we are using are at the smallest possible size for the ROM generator, and the amortized ROM overhead is relatively large.

We also analyzed the tradeoff between a full 256 entry gradient table described in Section 4.4 and a much smaller 64 entry gradient table which we believe is the smallest size that generates results that are visually indistinguishable from the larger table sizes. Because we replicate this table eight times to allow for parallel lookups, the area savings are considerable. We opted to shrink the table rather than pipeline the lookups in the table for simplicity and parallelism. In addition, this reduction in the size of the gradient tables allowed us to shrink the bit width of the 256 entry hash tables as we only need six bits to find the gradient. In each case, the gradient values used in our tables were generated using the point repulsion technique described in Section 4.4.

4.11 Pipelining

Our initial design was entirely combinational where it was assumed that a full computation of noise would be performed in a single cycle. We compared that design to a pipelined design with up to four stages and found that we could meet our goal of 1GHz frequency with four pipeline stages where the nonpipelined version would only reach 344MHz. The pipelined version was generated by first synthesizing the entire combinational circuit to meet the combinational timing requirement. Design Compiler was then run on the synthesized circuit to perform register retiming and distribute the registers throughout the circuit resulting in a pipelined implementation. Register retiming can create circuits of very different sizes depending on where the registers are placed in the final retimed version.

To explore the design space we synthesized a few different pipeline depths as part of our design process. The results of these synthesis runs are detailed in Table 4.1. While we were also able to achieve a 1GHz clock frequency with a three stage pipeline, the size of the three stage design was larger than the four stage design because a larger combinational circuit is needed to meet the timing requirements. We therefore chose the smaller design since throughput is more important than latency for this application.

Table 4.1. Cell area and performance of synthesized noise module. Determined from synthesis before place and route.

Pipeline stages	Clock (MHz)	Cell Area (μm^2)		
		Combinational	Sequential	Total
0 (combinational)	344	60350	0	60350
1 (not retimed)	337	57052	2470	59522
3 (retimed)	1000	76256	11980	88236
4 (retimed)	1000	58090	11470	69560

4.12 Physical Implementation

Figure 4.13 shows the computational flow of information for a single noise calculation. A three-space point (vector) is input and the result is a single noise value which is used in the shading computations. From this diagram the parallelism in the algorithm is apparent, and would be difficult to exploit in software. The thick lines in the diagram are vectors (typically three elements of eight bits each) and the thin lines are single eight bit values. The boxes labeled “Hash” and “Grad” are the hash lookups and the gradient tables described in Section 4.10. Section 4.6 also describes the polynomial operation (radial filter) performed by the “Poly” boxes.

The arithmetic implemented in the magnitude, dot product, and polynomial computations, as well as the multipliers and adders, is all fixed point. The range of values for the fractional inputs is in the range $[0, 1]$ and only the most significant bits really matter, which is why fixed point is sufficient and beneficial to our design. Our design retains all the bits needed to encode the exponent in a floating point number and also results in smaller circuits because of the fixed point representation. The arithmetic circuits were described with behavioral verilog and synthesized with Synopsis.

While this design was not fabricated, the final circuit after synthesis and place and route can be seen in Figure 4.15. The size of this final layout is $105\text{k}\mu\text{m}^2$ ($325\mu\text{m} \times 325\mu\text{m}$). For comparison, and also in the context of the TRaX processor, we produced other more well-known circuits using the same standard cells and the same 65nm technology. A single-precision floating-point three-element dot product takes an area of $111\text{k}\mu\text{m}^2$. A double-precision floating-point multiplier consumes $73\text{k}\mu\text{m}^2$, while a single-precision floating-point ALU (performing add, subtract or multiply) uses $34\text{k}\mu\text{m}^2$ in this technology.

Our circuit is smaller than a single-precision dot product, despite containing eight dot product operations because we use lower precision arithmetic in our circuit. While this reduced precision is not appropriate for every potential application of noise, we believe it is sufficient for our intended application of

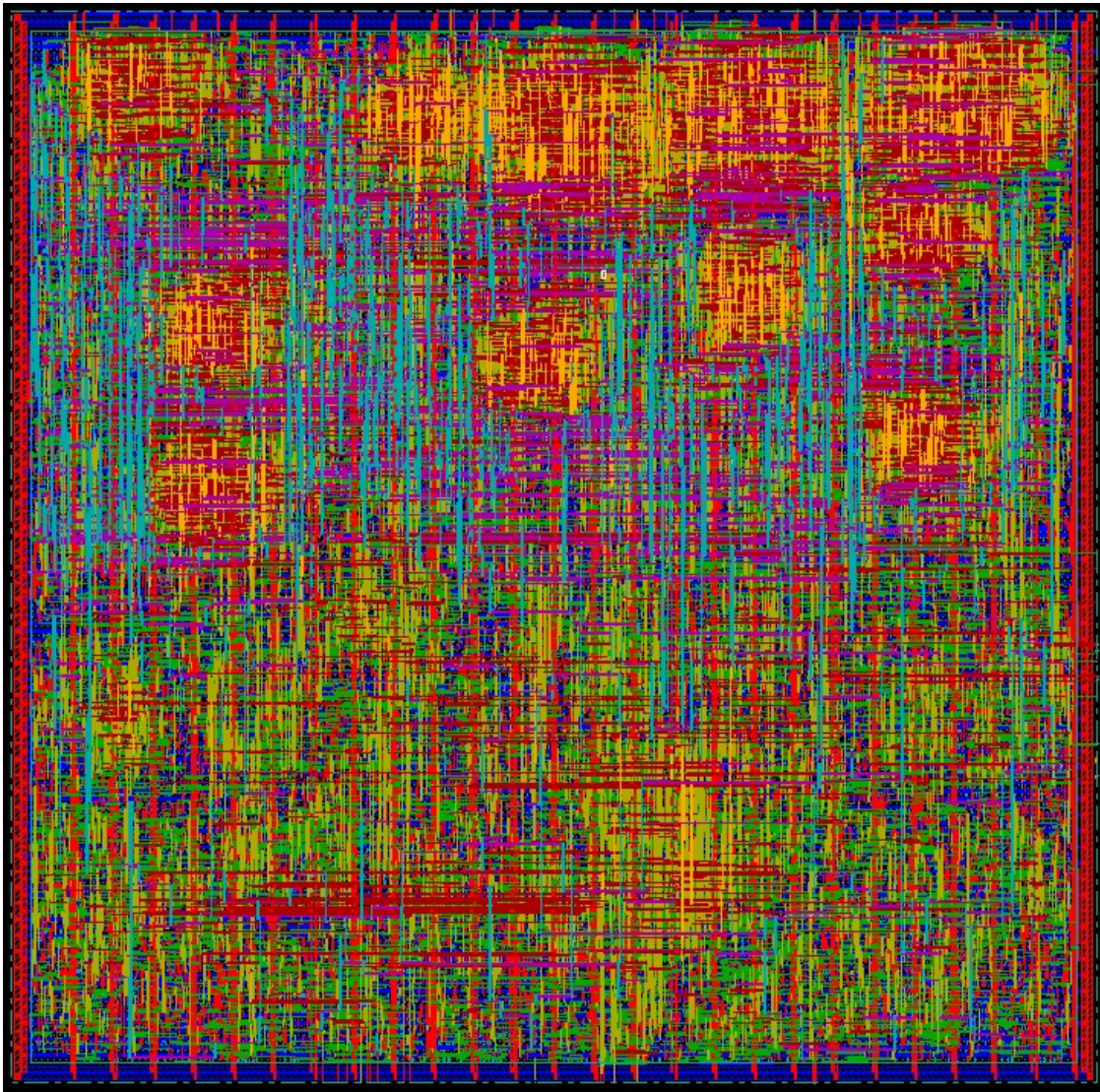


Figure 4.15. Placed and routed circuit implementing our improved noise. This image is a screen capture from Cadence SOC Encounter and shows only metal routing layers of the four-stage pipeline circuit ($105\text{k}\mu\text{m}^2$).

shading in graphics. It would also map well to any other application that ends up truncating the precision when using the results of the noise.

4.13 Conclusion

Our improved noise algorithm results in high quality noise that avoids the downfalls of periodicity, anisotropy, and aliasing. This functional unit performs this operation quickly and requires only a relatively small die area. We

reached our target of a 1 GHz clock frequency with a four stage pipelined design which produces one noise value per clock once the pipeline is full. This can be compared to a straightforward software implementation of Perlin noise which requires 120 floating point operations and peaks at 16.7M evaluations per second on a single core of 2.8GHz Core 2 Duo. Our final design uses three 256 entry hash tables where, to avoid additional adders, each table entry encodes the hash value for the input, and for the input plus one (see Figure 4.13). We also use eight copies of a 64 entry gradient table, where each gradient is a three-element vector of fixed point values.

As graphics pipelines demand more and more memory bandwidth we believe that providing a method for high quality textures through a hardware accelerated noise function provides a good trade-off. Much of the bandwidth of high-performance graphics chips is devoted to image-based (lookup) texturing. Procedural textures using noise offer an alternative that trades memory bandwidth for computation. The scene in Figure 4.1 is an example that uses an average of 552 calls to the noise function per shading sample. 37.2% of the total execution time for rendering the image was spent in the evaluation of noise for various aspects of the image. The textures on all of the surfaces and the smoke use noise to improve visual quality. The use of image-based textures would require far more memory bandwidth than our approach.

Admittedly, many applications would see more modest improvements in performance than the specific scene used here which is designed to demonstrate heavy use of noise-based textures. However, any time noise is used there would be a speedup using our hardware over a software implementation. At least one place where this could encourage visually complex images at a reduced memory bandwidth requirement would be video games. Games typically use very large image textures to avoid the appearance of repetition. While we do not have specific projections of memory bandwidth savings, it is well known that the large image textures are a significant fraction of the memory bandwidth in video games. Our design could increase the performance of applications that

use noise by as much as 50% and would be a good step toward high quality procedural texture generation and could become a viable real-time alternative to image-based texturing.

CHAPTER 5

IMPROVING BOUNDING VOLUME HIERARCHIES THROUGH TREE ROTATIONS

This work is based on an earlier work © 2008 IEEE. Reprinted, with permission, from Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing, Tree Rotations for Improving Bounding Volume Hierarchies, Andrew Kensler.

Current top-down algorithms for constructing bounding volume hierarchies (BVHs) using the surface area heuristic (SAH) rely on an estimate of the cost of the potential subtrees to determine how to partition the primitives. After a tree has been fully built, however, the true cost value at each node can be computed. We present two related algorithms that use this information to reduce the tree's total cost through a series of local adjustments (tree rotations) to its structure. The first algorithm uses a fast and simple hill climbing method and the second uses simulated annealing to obtain greater improvements by avoiding local minima. Both algorithms are easy to add to existing BVH implementations and are suitable for preprocessing static geometry for interactive ray tracing.

Bounding volume hierarchies are also a popular choice for ray tracing animated scenes, due to the relative simplicity of refitting bounding volumes around moving geometry. However, the quality of such a tree can rapidly degrade and thus some degree of rebuilding is still necessary. We show how animations can benefit by combining efficiently combining refitting with tree rotations. The result is a fast, lightweight, incremental update algorithm that requires negligible memory, has tractable update times and parallelizes easily, yet avoids significant degradation in tree quality or the need for rebuilding while maintaining low rendering times.

On average, our incremental tree-update takes slightly less than double the time for refitting alone while producing BVHs that are only $1.4\times$ more expensive to render than those produced by completely rebuilding with a high quality sweeping SAH algorithm for each frame.

5.1 Introduction

Bounding volumes [17] are a relatively simple idea: find a simple object – axis aligned boxes are currently the most popular – that completely contains, or bounds, a more complex object. If the bounding volume is not visible, then nor is the object that it contains. In the context of ray tracing, we perform a ray intersection test against the complex (expensive to intersect) object if the ray intersects the bounding volume. This reduces the overall computation if the rays miss enough of the time. Whitted’s original ray tracer [103] used bounding volumes this way. Rubin and Whitted [84] improved on this by organizing the bounding volumes into a hierarchical tree structure, the bounding volume hierarchy (BVH). Weghorst et al. [102] noted the importance of considering the projected “void area,” and proposed the idea of a cost model for selecting bounding volumes. Kay and Kajiya [46] introduced a top-down method for automatically creating bounding volume hierarchies from collections of objects by recursively sorting their medians along an axis and then subdividing them.

Goldsmith and Salmon [32] gave the basis for a cost function that is known today as the surface area heuristic (SAH), and used it to build BVHs from the bottom-up, but inserting new primitives into the tree one-by-one. MacDonald and Booth [60] applied the idea to spatial acceleration structures known as KD-trees [9], and used it in what is nearly its modern form [41] to choose splitting planes for the KD-trees. Their algorithm worked top-down and recursively divided the objects and space, greedily choosing the split plane that minimized the SAH at each level. Müller and Fellner [65] modified this for use in BVHs and gave a similar top-down SAH-based construction algorithm for BVHs. The algorithm scans along the list of primitives that have been sorted along an axis by

centroid and seeks to divide the list into two pieces such that the SAH function is minimal. Mahovsky [61] noted that the benefits from the surface area heuristic are similar for kd-trees and BVHs. SAH-based algorithms dominate the state of the art in construction algorithms for both acceleration structures.

The surface area heuristic provides a cost model for the average time that a ray will take to traverse down the tree and intersect candidate primitives at the leaf nodes. In its basic form as used for binary tree BVHs constructed over triangle meshes, it says that the cost, C , of a node is

$$C = \begin{cases} C_t + C_i N & \text{for leaf nodes} \\ C_t + \frac{S_L C_L + S_R C_R}{S} & \text{for interior nodes,} \end{cases}$$

where C_t and C_i , respectively, represent the relative costs for a traversal step and for a primitive intersection step in the rendering implementation and N is the number of primitives at a leaf node. S_L , S_R , S give the respective surface areas of the bounding volumes of the left child, right child and current node. Similarly, C_L and C_R give the computed costs for the left and right children.

The SAH algorithm for constructing BVHs proceeds in top-down fashion. At each node it considers a set of candidate partitionings of the primitives and greedily chooses the one that minimizes its estimate of the node's cost. Typically, "exact" SAH algorithms form these candidate partitionings by using the planes from each of the six sides of each primitive's axis-aligned bounding box. These partitionings appoint each primitive to a group based on which side of the plane the primitives' centroid falls on. Once the best partition has been found, these groups become the current node's children and then the process continues recursively down until the number of primitives assigned to a node falls below some threshold at which point that node becomes a leaf.

Approximate SAH construction algorithms for BVHs [96] work similarly, except that they use binning to sample a smaller number of potential partitions—often by simply choosing axis-aligned planes at regular intervals rather than the sides of the primitives' bounding boxes. From these, they fit a simple curve to the cost estimates and use the curve's minimum to choose the best plane along

which to partition the primitives. Thus, the approximate algorithms attempt to build a tree with a quality close to those of the exact algorithms' but in a fraction of the time.

One notable characteristic of all top-down construction algorithms based on the SAH is that they are not minimizing the true value of the C but only an estimate because C_L and C_R can only be computed with certainty after their corresponding subtrees have been fully constructed. Instead, they estimate C_L and C_R either linearly or logarithmically from the number of primitives potentially assigned to each child.

After fully constructing the tree, however, one can compute the true cost value both for the tree as a whole and for each node. With this additional information it may be possible to produce an improved tree with a lower cost. Ng and Trifonov [68] partially explored stochastic algorithms for this by repeatedly building BVH trees with jittered splitting plane locations, and then preserving the tree that yields the lowest true cost. This method failed to produce better trees than the standard greedy algorithm. Inspired by the use of genetic algorithms for constructing binary space partition (BSP) trees in [15], they also explored extending Goldsmith and Salmon's [32] bottom-up BVH construction method with a genetic algorithm to optimize the order of primitive insertion. Despite the improvements yielded by the genetic algorithm, they found that the standard top-down greedy build algorithm still produced superior trees.

In the first half of this chapter we explore an alternate method based on starting with an already existing BVH such as one produced by the usual top-down greedy algorithm and then using local tree rotations—similar to those used for balancing binary search trees—and hill climbing to improve it. Then we show how to combine this with stochastic global optimization via a simulated annealing algorithm [51, 92] to attempt to avoid local minima. Following that, we explore the application of tree rotations to maintaining up-to-date BVHs for animations.

Acceleration structure maintenance is a crucial component in any interactive

ray tracing system with dynamic scenes. As the geometry changes from frame to frame, the existing acceleration structure must be either updated or replaced with a new one. An ideal update algorithm should produce an acceleration structure that is as efficient to render as an acceleration structure produced by a high quality, offline build algorithm for the same frame, yet produce it in as little time as possible.

In the past few years, bounding volume hierarchies have been popular subjects of research for efficient update algorithms. They are relatively quick to render and there is a very simple update algorithm which uses refitting [97]. Refitting works by performing a postorder traversal of the nodes BVH tree. Each leaf is updated with a new, tight bounding volume over its corresponding geometry and then interior nodes combine these to form a tight volume enclosing their children. With a binary tree and axis-aligned bounding boxes, this process is fast and reasonably effective for small deformations to the underlying geometry. However, it can degrade rapidly when the geometry moves incoherently or undergoes large topological changes.

Full rebuild algorithms overcome this by replacing the BVH tree with a new one before the degradation becomes too significant. Lauterbach et al. [55] took the approach of measuring the degradation and performing the rebuild on demand. Ize et al. [42] created a system that continuously performs a rebuild asynchronously and substitutes the new tree on completion. Wald's [96] program completely avoids degradation by using a fast build algorithm to completely rebuild the tree for every frame.

Alternatively, hybrid algorithms combine refitting with heuristics to determine when to perform a partial rebuild or restructuring of a subtree. Yoon et al. [106], for example, implement a cost/benefit estimate of the culling efficiency of ray intersection tests to restructure pairs of nodes, while Garanzha's [31] algorithm looks for nodes whose children undergo divergent motion. Both of these algorithms use multiple phases to first identify candidates for restructuring and then to reconstruct them.

In the second half of this chapter, we describe a much simpler, yet still effective approach based on tree rotations. Combining refitting with tree rotations and produces a fast incremental update algorithm that achieves much of the simplicity and speed offered by refitting alone while attaining rendering performance closer to full rebuilding algorithms.

5.2 Tree Rotations

Self-balancing binary search trees such as AVL [1] and red-black [7] trees use tree rotations as the fundamental operation for re-balancing the tree after an insertion or deletion. Splay trees [89] also use tree rotations, but apply them after a lookup in order to improve future access times when it needs that element again soon.

Tree rotations are local operations involving the root of a subtree and its immediate children and grandchildren and come in two symmetric forms: left-rotations and right-rotations. By altering the linkage of nodes within the tree, one of the children moves up to take the place of the root of the subtree, while the original root descends to become a child of the new root (Figure 5.1). Which child ascends and which descends depends on whether a left-rotation or a right-rotation is being applied. In either case, a rotation will undo the effects of its opposite. Furthermore, rotations always preserve the search ordering of the original tree and it is possible to transform any two binary search trees with the same nodes into the each other through a sequence of rotations.

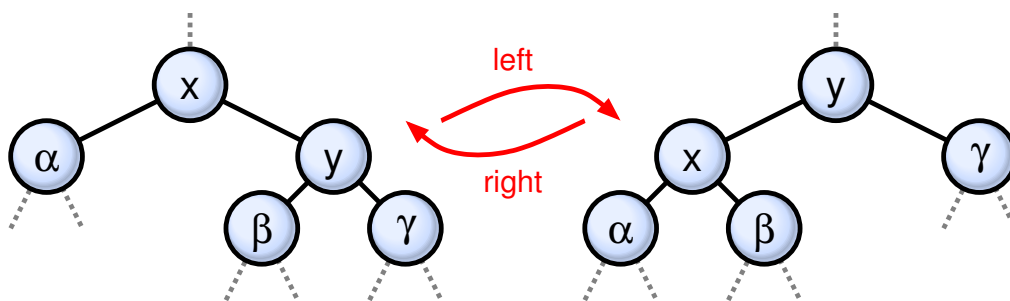


Figure 5.1. Left- and right-rotations on ordered binary trees

We can apply a similar idea to modifying BVHs. Because the BVH has no order property and the interior nodes carry no inherent information of their own—only the links to their children and the union of their children’s bounding volumes—the rotation operations are slightly different, however. The rotations reduce to simply swapping a child with a grandchild from the other side. Figure 5.2 shows the four possible pairs to exchange. As with the rotations for binary search trees these result in elevating one subtree, rooted at the grandchild, while demoting another, rooted at the child.

Note that these exchanges do not affect the bounding volume at the subtree’s root—only its children’s bounding volumes will change. Because of this, an exchange will not affect the bounding volume of any node in the tree above the subtree’s root and so the denominators in the computations of the SAH costs for these upper nodes will remain unchanged. Consequently, any rotation that increases or decreases the subtree root’s cost must produce a corresponding, monotonic change to the global cost of the BVH for the entire scene. Applying a rotation that improves the subtree’s cost will always improve the scene’s global cost and choosing the rotation that produces the greatest reduction to the local cost will produce the greatest reduction to the global cost. Moreover, computing the local effect of a rotation is a constant-time operation if the costs and surface areas of the children and grandchildren are available.

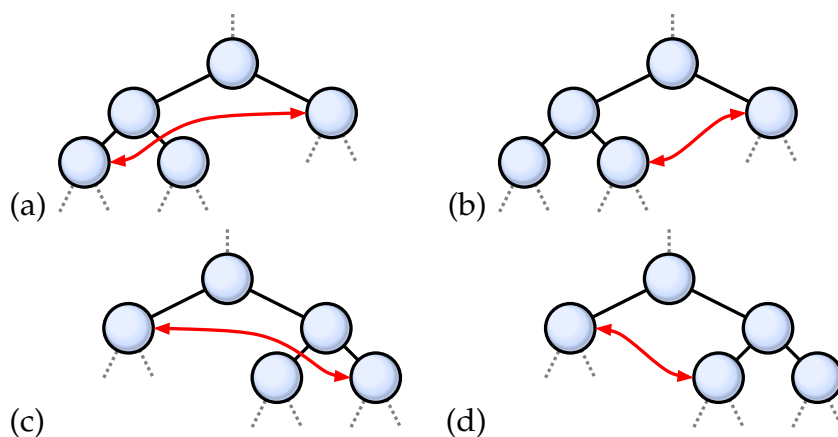


Figure 5.2. Nodes to exchange for each possible rotation

5.3 Hill Climbing

This leads to a very straightforward, efficient, hill climbing algorithm for improving the cost of a BVH after it has been constructed: recurse over the tree and visit each node. If the node is a leaf node, simply compute its cost and return. If the node is an interior node, visit its children first and then recompute its current cost. Next, determine which rotations the node is eligible to be the root for and compute the new costs that the node would have if the rotations were applied. If any of them yield an improvement over the current cost, then apply the rotation that produces the greatest improvement, update the node's cost and return. Otherwise, leave the node unchanged and return to the parent. Repeat these passes through the tree until no new beneficial rotations can be found.

One improvement that we have found beneficial is to allow for direct swaps between the grandchildren of a node. While not true rotations, the procedure for updating the tree and computing the effect of these swaps on the node's cost is nearly identical to that for true rotations. Though a sequence of rotations can produce the same effect, the intermediate steps may temporarily increase the tree's cost and thus the hill climbing algorithm would overlook them.

Figure 5.3 demonstrates this algorithm applied to the standard conference room scene. Starting with the initial global SAH cost after the tree's construction, each pass of our algorithm progressively lowers the cost until it cannot reduce it any further.

5.4 Simulated Annealing

While the algorithm described above will never increase the global SAH cost of a BVH it tends to quickly settle into local minima. To remedy this we used the simulated annealing algorithm [51, 92], a variant of the Metropolis algorithm [62]. Inspired by the annealing methods for growing large crystals, simulated annealing uses the concept of a global temperature to control moves that worsen the solution but also allow it to escape the minima.

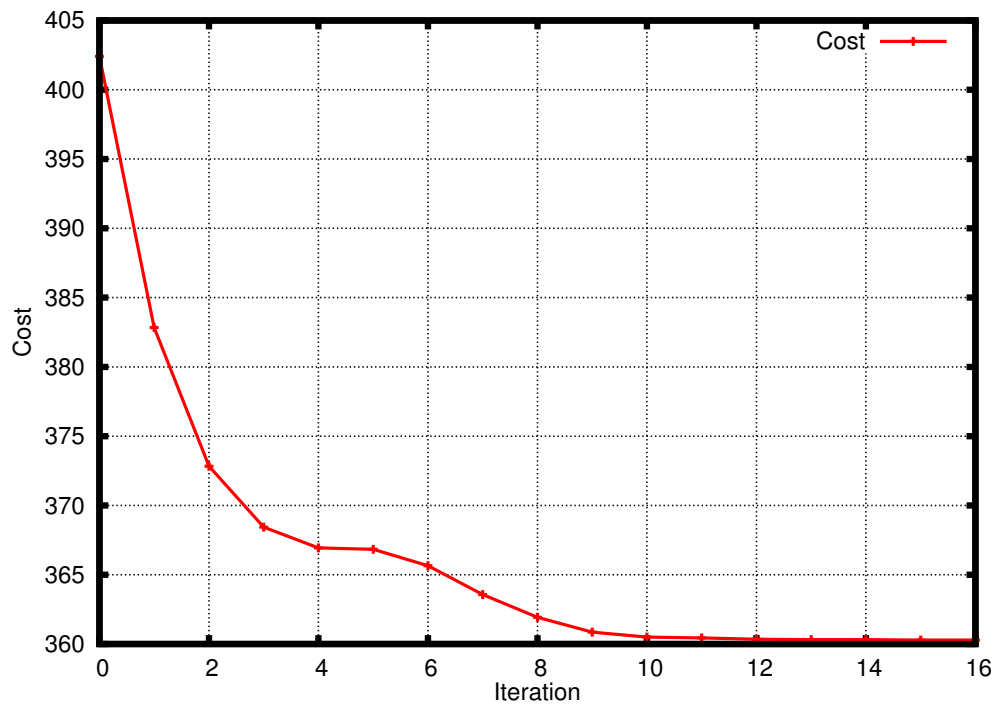


Figure 5.3. Progressive lowering of SAH cost for conference room

The algorithm requires several pieces: an energy function to compute the energy of a particular state, a way to propose candidate transitions to neighboring states, an acceptance probability function, and an annealing (cooling) schedule that defines how the temperature changes over time. The algorithm works by repeatedly proposing changes that the energy function measures the effect of. It always accepts changes that decrease the energy but for changes that increase the energy, the acceptance probability function uses the measured difference in energy and the current temperature from the annealing schedule to compute a probability. It accepts the proposed change if a random number in $[0, 1)$ is less than this probability and rejects them otherwise. After each iteration, it adjusts the current temperature according to the annealing schedule. Finally, there is often a “quench” phase after the annealing schedule ends. This phase moves to the local minimum by only accepting changes that reduce the systems energy.

Mapping this to the BVH tree rotation algorithm is straightforward. The SAH metric provides the energy function, while the set of rotations possible at

each node provides the proposed state transitions. For the acceptance probability function, we use the standard Boltzmann factor used by most simulated annealing implementations,

$$P(\Delta e, T) = \begin{cases} \min(e^{-\Delta e/T}, 1) & T > 0 \\ 0 & T = 0, \end{cases}$$

where Δe is the change in energy produced by the proposed change and T is the current temperature. Through experimentation we have found that a clamped and linearly ramped sine function makes a reasonably good annealing schedule. We used $T(i) = \max(0, -\sin(i2\pi/f))(N - i)h/N$, where i is the current iteration over the tree, f is the sine function's frequency, h is the hottest temperature allowed, and N is the schedule's length, defined as the total number of iterations to run. In our tests, we used $N = 1250$, $f = 50$ and $0.8 \leq h \leq 2.2$, depending on the scene.

To implement this, we start with the previous hill climbing algorithm that recurses over the tree and at each node test each possible rotation and swap to determine which one lowers the local cost of the node the most. However while trying to find the minimum if a proposed exchange would increase the node's local cost relative to the currently best found exchange then instead of outright rejecting it as before, we pick a random number ξ in $[0, 1)$ and test if $P(\Delta e, T) < \xi$ and use the difference in local cost as Δe . If this test succeeds then we accept this as the currently best found exchange anyway. After all valid rotations and swaps around a node have been tested this way, we apply the best one, if any. After the recursion has finished processing all the nodes this way, we update the temperature for the next pass and begin walking the tree again. Because we define $P(\Delta e, T)$ to always be zero when $T = 0$, the simulated annealing algorithm reduces to the original hill climbing algorithm for the phases of the annealing schedule when $T = 0$. At the end of the schedule, we quench the system by leaving $T = 0$ and continue to make passes until the global cost of the BVH tree converges to a local minimum. Optionally, we can retain a copy of the best tree found as the optimization progresses and replace it whenever

a pass ends with an improvement to it, then return that tree as the simulated annealing algorithm's result. If the annealing schedule starts out with $T = 0$ for enough iterations with this option then in the worst case the simulated annealing algorithm will at least match the hill climbing algorithm.

Figure 5.4 shows how the simulated annealing algorithm reduces the global SAH cost for the Soda Hall scene. The annealing schedule begins with a phase where the temperature is zero and so it will initially lower the global cost for the scene as with the hill climbing algorithm before. After a few iterations, the temperature rises and the algorithm makes changes to the tree that result in increasing the tree's cost. By cycling through these heating and cooling phases while gradually diminishing the strength of the heating phases, the BVH for the scene is brought to a lower global SAH cost than would be possible with the hill climbing algorithm alone (left plateau on the graph.)

Note that for any two trees with the same leaves and the same number of interior nodes, there is a sequence of rotations that will transform one into the

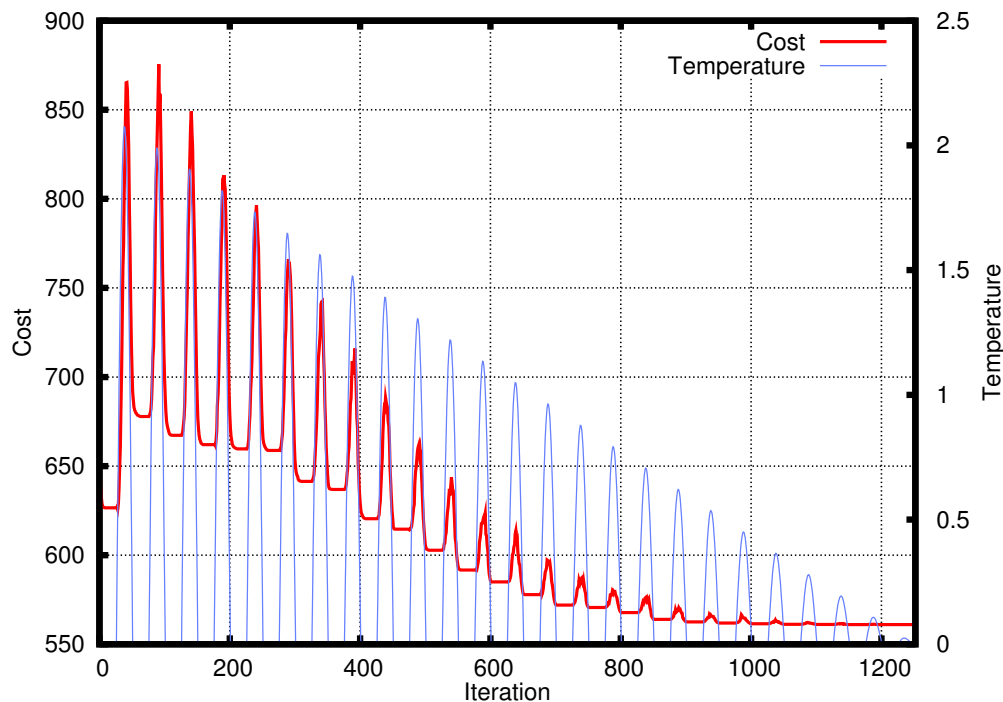


Figure 5.4. Simulated annealing applied to the Soda Hall model

other. Because the number of leaf nodes determines the number of interior nodes, the best possible BVH (as defined by global SAH cost) for a given set of leaves is always reachable from any other state. With this property, the probability of the simulated annealing algorithm coming across the best possible BVH approaches one as the annealing schedule grows [3]. While this will require an infeasible amount of computation in practice, the algorithm is nonetheless theoretically sound.

5.5 Performance on Static Geometry

We have implemented our optimization algorithm in the Manta interactive ray tracer [10]. The BVH code in Manta evolved from a port of the code from the DynBVH system [97]. We have kept the existing BVH construction and traversal code otherwise unmodified and simply hooked in our tree optimization to run after the BVH's initial construction. For all tests, we used the exact SAH build algorithm rather than the approximate binning algorithm.

For interior nodes, the BVH traversal code tries to improve the chances of early ray termination by choosing which child of an interior node to process first based on the split axis recorded for that node and the ray direction's sign along that axis. Because the optimization process effectively destroys this information, we have added a final pass over the BVH tree to try recreate this. At each interior node, this checks the bounding box of the children and determines which axis provides the greatest separation between the end of the one child's extent and the beginning of the other child's extent. It records this as the node's split axis and the orders the children accordingly. If there is no such axis that separates the bounding boxes' extents then it uses the axis with the least overlap of the extents instead.

We benchmarked the performance impact of the changes on rendering with a single core on a 2.8GHz Core 2 Duo machine so as to avoid jitter and overhead from threading synchronization. For our tests for static geometry, we used two different rendering modes: first, we tested with pure ray casting (primary rays









only), traced in packets of 8×8 rays each and with the SSE code paths and culling via interval arithmetic [97] enabled. This represents state of the art SIMD packet traversal with highly coherent rays. To test the opposite end of the spectrum, we also measured the performance on two-bounce path tracing with the SSE code paths turned off, packets of a single ray each and 1024 samples per pixel. This test emulates classic single ray rendering with rays diverging to traverse the scene from many directions.

Table 5.1 shows the results of these test on several common models along with the times required by each algorithm to optimize them. The rows showing ray cast and path trace times list the average time in seconds needed to render a single frame. In both cases, the percentages indicate the time needed to render with optimization relative to the time to render without optimization. Similarly, the percentages given in the rows for the SAH cost compare the global SAH cost for the scene after optimization to the cost before. Thus for all rows lower percentages indicate greater benefits from the optimization. The table also provides the time each method spent on optimizing the trees.

As shown in Table 5.1, the simulated annealing algorithm required significantly more time to process the BVH tree than the hill climbing algorithm but achieved a better reduction in the scenes' SAH costs. The number of iterations to run the simulated annealing is a user controllable parameter, however, and thus it is possible to control the trade-off between the processing time and the degree of improvement to the tree.

Several things are noteworthy about these results. First, scenes with densely tessellated models such as the Happy Buddha, the Dragon, and even to some extent the Fairy Forest, present the greatest difficulty with respect to optimization. While it is still possible for the optimizer to find small improvements that lower the SAH cost of these scenes, the improvements made tend to be quite small and may even be slightly detrimental to the rendering times. The majority of the iterations of the simulated annealing algorithm for these scenes ended with the computed SAH cost of the scene greater than it was initially.

Table 5.1. Results of tree rotations on static scenes. All times are in seconds and all percentages are relative to the nonoptimized values.

Scene	Triangles	Optimize (s)		Costs		Ray Casting (s)		Path Tracing (s)				
		Hill	Sim.	None	Hill	None	Hill	None	Hill	Sim.		
	1087474	1.71	528.3	552.6	549.6	549.6	0.262	0.263	0.263	153.8	154.8	154.8
	871306	1.23	426.5	486.0	484.3	484.3	0.496	0.496	0.496	197.9	199.5	199.5
	121862	0.18	44.4	154.3	143.2	136.1	0.165	0.162	0.160	149.2	141.2	127.5
	365970	0.53	114.1	278.8	260.4	247.3	0.365	0.363	0.354	572.1	567.9	524.2
	174117	0.24	74.1	294.9	290.2	281.6	0.372	0.373	0.374	370.1	376.2	368.6
	76065	0.13	28.5	713.5	647.6	600.8	0.294	0.307	0.315	565.5	561.4	547.0
	282664	0.68	119.3	402.4	360.3	340.2	0.229	0.224	0.216	504.4	446.8	415.7
	2169132	4.93	857.6	685.5	626.6	561.0	0.326	0.320	0.306	1436.8	1417.1	1403.4

Interestingly, this suggests that the top-down greedy construction algorithm succeeds at producing good trees for these scenes despite using only an approximation to the true cost of the subtrees. At the very least, these trees seem to be close to strong local optima if not the global optimum. This confirms that the greedy approach and the approximations to the subtree costs are valid approaches for these models.

For scenes with triangles of heterogeneous sizes such as the architectural scenes, the optimization seems to fair much better. Here, the simulated annealing algorithm is able to achieve a 15.5% reduction to the cost of the Conference Room scene and a corresponding 17.6% improvement to the path trace rendering time. For these types of scenes, the top-down greedy build does not come as close to building trees near the local optima as it does for the finely tessellated scenes and thus the tree optimizations succeed better at lowering the cost.

Also notable is the degree of correlation between the changes in the SAH cost and the changes in the rendering times. In particular, the correlation between the cost value and the time for packetized ray casting appears to be much weaker than it is between the cost value and the time to path trace with single rays. While this is not surprising, as the surface area heuristic was developed based on the assumption of tracing single rays with well distributed directions, this does suggest the need for an improved heuristic that takes into account the amortization afforded by ray packets with culling. We believe that this would be a valuable area of future work.

5.6 Update Algorithm

The core of our incremental update algorithm for dynamic scenes is a straightforward combination of refitting and tree rotations, with the two intertwined into a single pass. For each frame, the scene geometry is first interpolated to its new position. Next, a postorder recursive pass refits the bounding box for each to enclose the geometry in its new position. After finding the new bounding box for a node the algorithm checks the children and grandchildren for possible

local exchanges of nodes (“tree rotations”) that might reduce the SAH cost of the subtree and thereby improve the overall quality of the scene BVH.

Figure 5.2 showed the four tree rotations that the algorithm considers. Each of the four rotations are primitive and involve exchanging a direct child of the node with a grandchild on the opposite side. This has the effect of raising one subtree at the expense of lowering the other. It also considers the two compound moves – direct swaps between “cousins” – that can be composed through sequences of the upper basic four rotations. Adding in these two compound moves, however, gives the tree rotation algorithm freedom to find improvements that it might miss due to the intermediate steps raising the SAH cost of the node.

For each node, the algorithm always greedily performs the rotation that minimizes the local cost of the node. If a rotation is found that reduces the cost, then the exchange is made and it refits the bounding box of the affected children and recomputes their new costs. Otherwise, if no rotation reduces the existing cost then the nodes are left as-is. This is essentially a single step of the basic hill-climbing algorithm described above.

One subtlety of our algorithm is in the recomputation of the SAH cost at each point. To decide on a rotation, the algorithm needs to compare the new cost of an interior node after refitting with the potential costs after each of the six possible rotations. Computing these costs involves dividing by the surface area of the node, but we can defer this division until after the selection of a rotation. Moreover, expanding out the cost formulas for each case leads to a number of common terms which help to reduce the expense of these cost computations.

Secondly, note that computation of the SAH costs requires no additional heap storage when updating the entire tree on each frame; evaluating a node for tree rotations requires the costs only at the children and grandchildren. With care, this information can be maintained entirely on the stack.

This algorithm is also easy to make faster through parallelization for multicore systems. Because the modifications to the tree structure are local and

incremental, we can divide the work up near the root of the tree and assign subtrees to each thread. Each thread performs a single pass over its subtrees to refit the bounding boxes, compute the costs, and perform any rotations. After a barrier, one thread handles this for the remaining nodes at the very top of the tree.

The biggest caveat of implementing our algorithm involves the handling of the primitives and the leaf nodes during the initial construction of the tree. Primitives whose motion may diverge need to be placed in separate leaf nodes. In practice, this means that our implementation builds the initial tree down to the level of a single primitive per leaf node, leading to larger trees. A smarter construction algorithm with a priori knowledge of how primitives move together [37] could improve on this.

5.7 Performance on Animations

As with the implementation of tree rotations for static geometry, we evaluated the performance of our update algorithm by implementing it in the Manta interactive ray tracer [10]. Starting with the existing recursive refitting code, we extended this to also maintain cost evaluations and to perform the most beneficial tree rotations as each node is visited. As before, we benchmarked it on a 2.8GHz Core 2 Duo, ray casting a 1024×1024 image using 8×8 packets. However, we ran Manta with two rendering threads this time.

We measured the performance on each animation for three cases: refitting-only, refitting with tree rotations, and doing a complete rebuild on each frame. Comparing against refitting-only gives an idea of the lower bound on the time to update the scene for each animation. This case produces very fast tree updates at the expense of the render time degradation as the animation progresses. At the other extreme, by performing a full, from-scratch rebuild of the tree using a high quality sweeping SAH construction algorithm [97] on every frame we get an idealized measurement of how low the render time could be if there were no degradation at all.







To calibrate the comparison between the different methods, we ran each animation with a fixed step size between frames, regardless of the actual wall-clock time required to update the BVH tree and render it. Each scene was run for 100 frames except for the Toasters scene which was run for 250 frames.

Table 5.2 shows the statistics for a set of animations. Each number, except for the triangles in the scene, is taken as the average over all frames during five runs. From left to right, this shows the number of triangles in the scene, average number of rotation operations per frame, and average time spent updating the tree using the refitting-only and refitting with tree rotation methods, followed by the average time spent rendering the updated frame after using the refitting-only update, full rebuild, and refitting with tree rotation methods. The last two numbers give the the average frame rates for updating and rendering with the refit-only and refitting with tree rotation algorithms.

Figures 5.5 and 5.6 show more detail on the relative performance of the DragBun scene. This scene is interesting for having two phases: it begins with the geometry moving very coherently and follows this with extreme topological deformations. The first graph compares the three update methods by the render time for each frame. It also shows the SAH costs of the frames for each method. The second graph goes into more depth on the refit with tree rotations algorithm, and shows the amount of time spent on refitting, cost evaluation and tree rotations, and rendering, along with the actual number of tree rotations performed.

As shown, adding tree rotations to refitting less than doubles the typical update time for a tree, while avoiding most of the tree degradation inherent to refitting alone. The ClothBall scene, which demonstrates relatively coherent motion is the only scene for which the addition of tree rotations to the update process is detrimental to the overall performance. Other scenes show modest improvements at worst, and the scenes with drastic deformations show significant improvements – almost ten-fold in the case of the hard body Balls16 simulation scene.

Table 5.2. Results for tree rotations on animations. Rotation counts, times, and framerate taken as averages over five full animation runs.

Scene	Triangles	Rotations	Update Time (s)		Render Time (s)		Framerate (fps)		
			Refit	Rotate	Refit	Rotate	Refit	Rotate	
 Toasters	11141	675	0.0010	0.0023 (2.28×)	0.0856	0.0752	0.0755	11.55	12.86 (1.11×)
 Balls16	146480	744	0.0142	0.0256 (1.81×)	1.6824	0.1047	0.1544	0.58	5.55 (9.42×)
 BART	65536	10757	0.0126	0.0203 (1.60×)	2.2687	0.2358	0.5494	0.44	1.75 (4.01×)
 ClothBall	92230	2386	0.0095	0.0168 (1.77×)	0.0778	0.0704	0.0767	11.46	10.69 (0.93×)
 DragBun	252572	5916	0.0289	0.0503 (1.74×)	0.9629	0.0637	0.0986	1.01	6.71 (6.66×)
 Fairy	174117	4515	0.0232	0.0436 (1.88×)	0.2436	0.2164	0.2224	3.75	3.76 (1.00×)

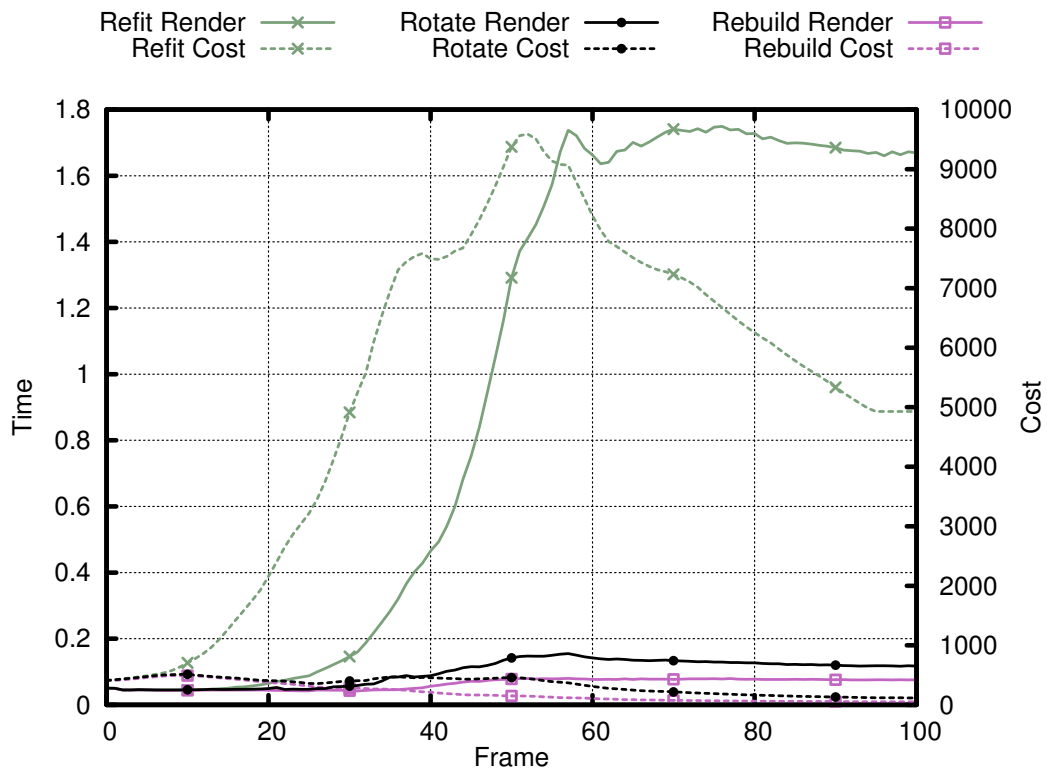


Figure 5.5. Render times and SAH costs on the DragBun animation. This compares refitting only, refitting with tree rotations, and completely rebuilding for every frame.

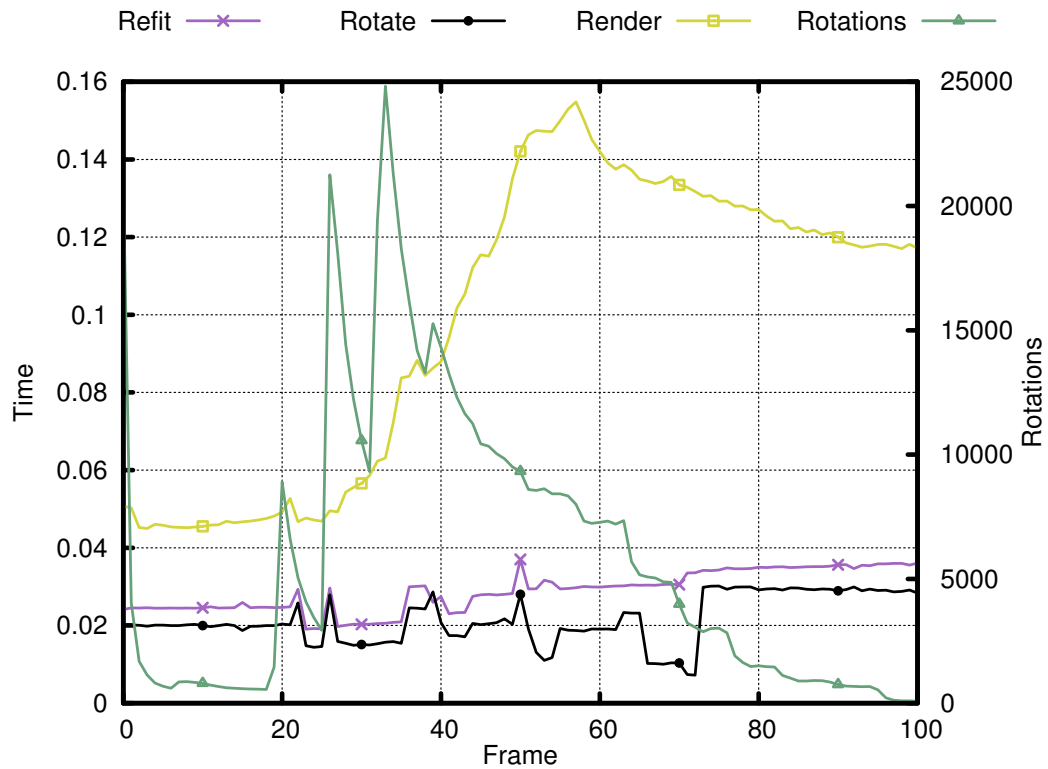


Figure 5.6. Break-down of time spent on the on the DragBun animation. This shows the time spent each frame on refitting, performing tree rotations and rendering on the DragBun scene. The number of tree rotations performed for each frame is also shown.

Part of the low overhead of adding refitting comes from the simplicity of the algorithm. Each refitting and tree rotation operation is simple, uniform and local. Because the combined algorithm accomplishes everything in a single pass over the tree, the penalty of cache misses is amortized between the refitting and tree rotation parts.

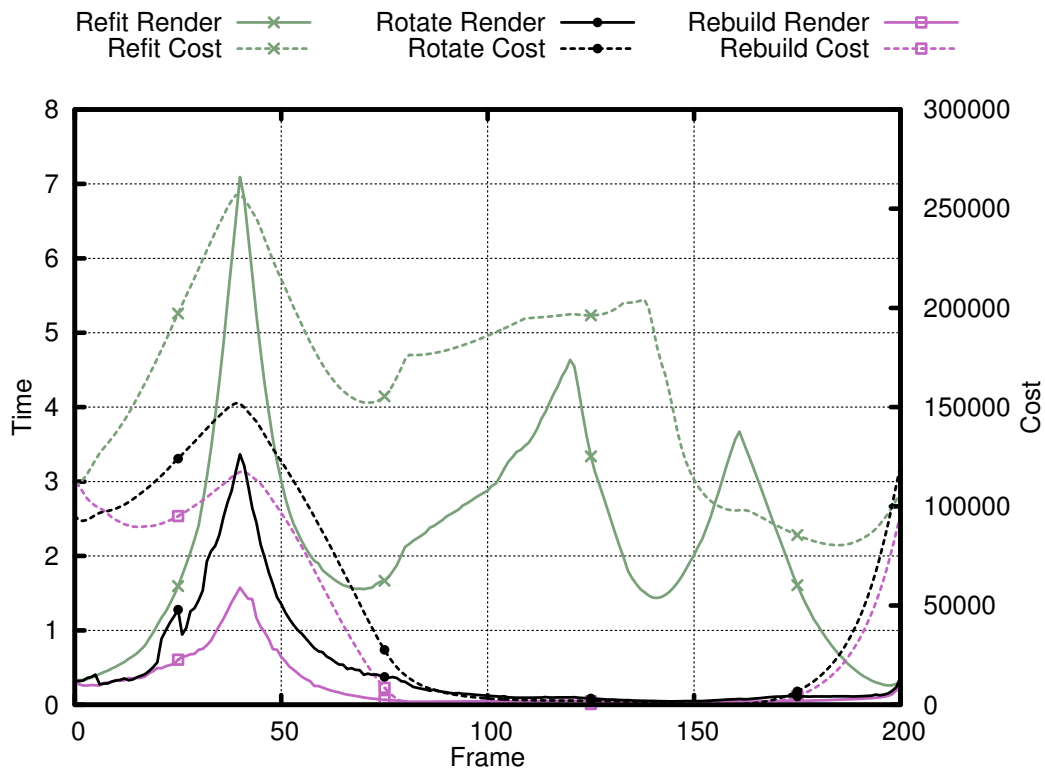
A second aspect of the low overhead is that only the interior nodes (amounting to half the nodes) require the full SAH formula to evaluate their costs. Moreover, only nodes with grandchildren can be candidates for tree rotations, reducing the number still more to a quarter of the nodes. Of these, the rotations statistic in Table 5.2 shows that only a small percentage of the nodes actually require rotation operations on any given frame – the BART museum scene requires the most with 8.2% of the nodes per frame on average. Most other scenes show closer to 1%.

One important question is how the update algorithm handles rapid scene changes. Because interactive animations typically choose an update step based on wall clock time, the update algorithms may have to cope with large steps. To test this, we examined the render times for the three algorithms on the BART museum scene with both long and short animations over 200 and 20 frames, respectively. Figure 5.7 shows the relative performance of the three methods in this case.

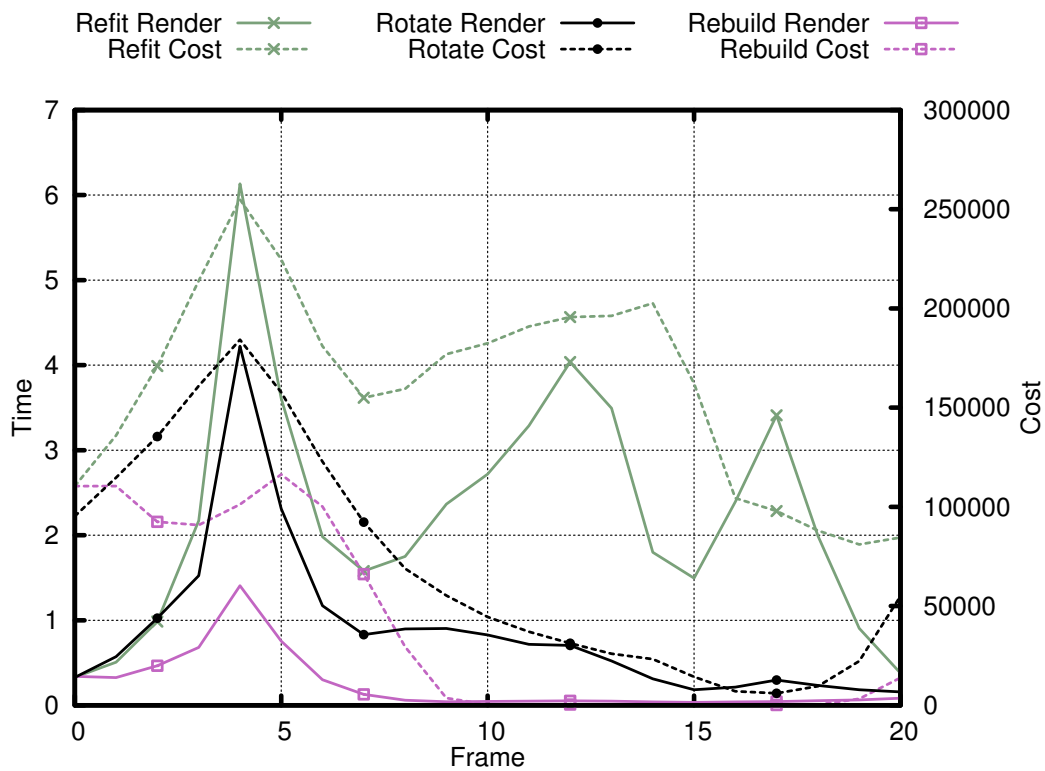
In this case, the rendering performance and quality of the tree does degrade somewhat. However, as soon as the animation reaches a more quiescent period the combined algorithm begins to catch up.

5.8 Modifying the Scene Graph

We envision the tree rotation update algorithm as a core component in a larger scheme for efficiently maintaining a high quality BVH acceleration structure in correspondence to a changing scene graph. The combination of refitting with tree rotations handles the case of updates for objects in the scene graph undergoing transformations, deformations or other movement.



(a) Long



(b) Short

Figure 5.7. Comparison on BART scene with long and short animation loops. Slower animations give more time to optimize the tree.

To handle the addition of new objects to a scene graph, each potential object could have its own small prebuilt BVH tree for the object in a basic rest pose. When the object is added to the scene graph, we would make a complete copy of the object's BVH and geometry, in the process applying any transformations and placing the geometry in world space if necessary. Next, the root of this copied BVH would be anchored next to a leaf in the existing scene BVH in the manner of [32]: beginning at the root of the scene BVH, walk down to a leaf, always choosing the child whose costs would increase the least as a result of the addition. This would provide a reasonable initial change to the tree, after which refitting and tree rotation would update the copied bounding boxes and begin better integrating the new object into the scene's BVH.

Deletion is slightly more difficult. To delete an object, perform a postorder traversal of the tree. At each leaf node, check to see whether it corresponds to a scene graph object flagged for deletion. If so, overwrite its parent node with its sibling's node data and move the leaf and its sibling to a free list for later reallocation. Effectively, this always deletes a leaf node together with an interior node (thus deleting the same total number of interior and leaf nodes as originally inserted for the object). However, the actual memory reclaimed is of a pair of sibling nodes, allowing implementations that allocate siblings in pairs to also recycle them together. Note that these deletion steps can be combined with refitting and tree rotations into a single pass for efficiency on frames where objects must be deleted. Following this pass, the memory for the object's geometry can be freed and the object removed from the scene graph.

Ultimately, we believe that these techniques together would be sufficient to incrementally maintain a high quality BVH tree for interactive rendering, particularly in the case of games or other systems that allow the user to manipulate the scene. The relative simplicity of our proposed techniques also makes them candidates for implementation on a GPU or custom ray tracing hardware where maintaining an up-to-date acceleration structure in device memory could significantly reduce host-to-device bandwidth.

5.9 Conclusion

In this chapter, we have presented two novel algorithms for improving the SAH cost of an existing static BVH tree. Both algorithms are easy to add to existing BVH implementations and are suitable for preprocessing the static content of scenes before rendering, with architectural environments seeing the best improvement. The hill climbing algorithm can yield modest improvements to a scene with millions of primitives in a few seconds, while the simulated annealing algorithm can produce greater improvements given more time.

We have also described a simple and easy to implement technique for maintaining high quality BVH trees for dynamic animations based on a combination of refitting with tree rotations. This lightweight approach is efficient and incremental, uses a tractable amount of time to update the tree and requires no additional heap storage – properties that also make it excellent for parallelization. Our algorithm more than compensates for the increase to the tree update time with its improvements on render time, tending towards the quality of the trees produced by a complete tree rebuild on every frame. Together, these qualities make it a desirable alternative in any case where refitting alone is currently used.

CHAPTER 6

CONCLUSION

In this work we have provided contributions to three major areas related to interactive ray tracing on custom hardware: primitive intersection with ray-triangle intersection tests, shading with procedural texturing, and acceleration structure construction and maintenance with bounding volume hierarchies.

In Chapter 3 we showed how the mathematical structure of direct 3D ray-triangle intersection tests can be exploited by stochastic optimization algorithms to derive improved algorithms and higher performing implementations on modern processors for packet based ray tracers.

The new intersection test created with the assistance of our genetic algorithm has already improved the efficiency of at least one software ray tracer: it is one of the two triangle intersection algorithms available in the Manta interactive ray tracer [10]. Compared to the other test in Manta, Wald's 2D point-in-triangle algorithm [95] it is slower on static scenes that allow for the preprocessing that Wald's algorithm requires. For animations, however, where the geometry is changing from frame to frame, the lack of precomputation makes our algorithm the faster of the two.

We also showed how the structure of the ray-triangle tests could be exploited for a slightly different purpose: the derivation of a highly robust algorithm. Although we did not use a stochastic search algorithm to derive this test, collection of a large set of test data and an exhaustive automated search through the possible expressions was crucial to determine which expressions were the most stable. We also performed an automated sensitivity analysis to suggest that a "three-quarters precision" floating point type may be suitable for custom

hardware for ray tracing. With regard to software ray tracing, we believe that our robust algorithm is ideal for production batch renderers for which artifacts are unacceptable.

Chapter 4 deconstructed Perlin's noise algorithm and explored how each component – the gradient table, the hash function, and the filter kernel – contributed to its spectral qualities in the frequency domain. Based on this analysis we were able to incrementally modify the algorithm to produce a new version with improved visual quality.

Using a larger table of well-distributed gradient vectors avoided certain biases that appear as artifacts in the form of runs. Using a slightly different hash function with a table for each dimension instead of a single table used for all dimensions avoids the effect where each row of noise is repeatedly regularly but with a permuted shift, producing a strong striation on the Fourier transform. The third major change replaced Perlin's separable reconstruction filter with a new radial one. This produced tighter bandlimits on the noise's spectrum.

We also showed how a simple projection technique could be applied gradient noise to rectify the defect shown by Cook and DeRose [19]. This projection technique preserves our gradient noises improved spectral qualities when sampling on a surface embedded in a higher dimensional noise. For typical surfaces textured with a 3D volume of noise, this reduces a "splotchy" appearance that traditional 3D Perlin noise exhibits. Together, these changes reduce the appearance of structured artifacts leading to higher quality images for software ray tracing with procedural texturing.

With respect to hardware ray tracing, we also demonstrated how certain of our changes, such as the improved hash function and the way we compute the radial filter, actually increase the possibilities for fine grained parallelism. Based on this, we designed and synthesized a hardware noise unit that can evaluate noise with a short, three-cycle pipeline at a 1GHz clockrate. Our unit is designed to be embedded as a functional unit in a larger hardware ray tracing system. By replacing image texturing with procedural texturing where possible, a hardware

ray tracer could reduce some of its memory bandwidth.

In Chapter 5 we showed how tree rotations when applied to a bounding volume hierarchy took the form of local swaps between the nodes. Thanks to the recursively defined surface area metric we could quantify the relative effectiveness of each possible tree rotation around a node with respect to the estimated cost to trace a random ray through the BVH.

Using a simple hill-climbing algorithm to choose and apply tree rotations we were able to produce reasonable improvements to trees produced by a high quality SAH-based BVH construction algorithm. This established that the common greedy build algorithm while good, was not globally optimal.

We also tested stochastic optimization of the BVH by using simulated annealing instead of hill-climbing. While taking much longer to optimize a tree, this produced greater improvements. Both software- and hardware-based ray tracers could benefit from this algorithm by using stored, prebuilt, preoptimized trees for the static geometry in a scene.

Finally, we explored the application of tree rotations to dynamic geometry. By combining a fast implementation of a single step of the hill-climbing algorithm with refitting we were able to create a simple BVH update algorithm that avoided most of the degradation associated with refitting alone. Subtrees can be processed independently, making it easy to parallelize, and it can be implemented without additional heap storage which should make it particularly well suited to implementation on a programmable hardware ray tracing system such as TRaX. Maintaining an up-to-date acceleration structure on the ray tracing device should also help with reducing memory traffic between the device and its host.

6.1 Subsequent Work

Following the publication of our fast ray-triangle algorithm, Reshetov [82] has since compared against this algorithm and used the testing harness as a basis for measuring his own work on culling triangles to obtain even faster average intersection times.

Since the original publication our noise algorithm, Lagae et al. [54] have explored many of these ideas and produced a new noise algorithm that offers even finer spectral control over the noise function. It is not yet clear how their noise function compares to ours in terms of performance, however.

The research on tree rotations for static model BVHs recieved some discussion in the most recent issue of Ray Tracing News [39]. The discussion concerned how it would handle two particularly tricky test cases.

6.2 Future Work

The genetic algorithm framework described in Chapter 3 is fairly general. One interesting area to explore would be modifying the genetic algorithm to generate Verilog code to synthesize fixed function hardware units for ray-triangle intersection. As with the software implementation, maximizing speed could be one goal for the system. A potentially more interesting goal, however, would be minimizing chip area.

While ray-triangle intersection test are certainly an important component, ray-box test are also quite useful due to the prevalence of axis-aligned bounding boxes. These are the most common bounding volumes in BVHs and even a system that uses a different acceleration structure may use boxes as an initial test before complicated primitives. Unlike triangles with the scalar triple products, boxes do not lend themselves to quite the same variety of equivalent intersection tests. Nonetheless, a genetic algorithm could prove useful to determine how to schedule the computations and which early exit tests should be used where.

Similiarly the effect of reduced precision floating point on ray-box tests is worth exploring. Mahovsky's dissertation [61] discussed reduced precision BVHs at length, including a compact node representation and using low-precision integer arithmetic for the ray-box test. To our knowledge, no one has yet quantified the effect that different exponent and significand widths have on ray-box tests.

With regards to procedural texturing, the noise algorithm is but one component. Typically Perlin noise is called repeatedly in spectral summation loops. It

may be worth exploring to see how to compute the results of these higher level loops more efficiently. The work of Lagae et al. [54] is a good step in this regard but spectral summations can still handle a wider range of texture frequencies.

Similarly, a higher level procedural texturing hardware unit that manages one or more noise units to efficiently compute the results of these summations with filtering for antialiasing may be worth pursuing. After setting up the parameters, such a unit might generate a variety of complex textures such as marble, wood, granite or sky.

With regards to acceleration structures, we envision our tree rotation algorithm as one element of a complete system for maintaining a BVH acceleration structure in the face of an ever changing scene graph. While tree rotations have been shown to work for handling deforming geometry, extending the system to handle additions and deletions remains to be done.

Testing the dynamic tree rotations algorithm on a highly parallel system such as TRaX would also be a worthwhile test both of the algorithm and of the TRaX architecture.

A rather different application for the dynamic tree rotation algorithm could be for distribution ray tracing with motion blur. Fatahalian et al. [27] observe that partitioning the time into intervals and rendering in multiple passes allows for tighter bounds on the geometry within each interval. The tree rotation update algorithm for animated scenes should also work as a more efficient way to update the acceleration structure between intervals.

Lastly, from the tree rotations work on static scenes we noticed that a lower surface area metric did not always correlate to lower rendering times in practice for packet-based ray tracing. The assumptions made by the BVH version of the surface area heuristic are based mainly on single-ray behaviour. This raises the question of how the surface area metric might be improved on to better model the behaviour of modern packet style ray tracers. One very interesting project would be to use a genetic algorithm to explore alternate cost functions to control the construction of the BVHs. The fitness of each candidate would

be evaluated by using it to construct a BVH for a scene and then measuring the time that it takes to render with that BVH. This might lead to empirically derived alternatives to the standard surface area metric that map better to how state of the art packet-based ray tracers actually behave in practice.

REFERENCES

- [1] ADELSON-VELSKY, G. M., AND LANDIS, Y. M. An algorithm for the organization of information. *Soviet Mathematics Doklady* 3 (1962), 1259–1262.
- [2] AMANATIDES, J., AND CHOI, K. Ray tracing triangular meshes. In *Proceedings of the Western Computer Graphics Symposium* (April 1997), pp. 43–52.
- [3] ANILY, S., AND FEDERGRUEN, A. Simulated annealing methods with general acceptance probabilities. *Journal of Applied Probability* 24, 3 (September 1987), 657–667.
- [4] APODOCA, A. A., AND GRITZ, L. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers, San Francisco, CA, December 1999.
- [5] APPEL, A. Some techniques for shading machine renderings of solids. In *Proceedings of the Spring Joint Computer Conference* (April 1968), pp. 37–45.
- [6] ARENBERG, J. Re: Ray/triangle intersection with barycentric coordinates. *Ray Tracing News* 1, 11 (November 1988).
- [7] BAYER, R. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica* 1 (December 1972), 290–306.
- [8] BENTHIN, C. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, 2006.
- [9] BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (September 1975), 509–517.
- [10] BIGLER, J., STEPHENS, A., AND PARKER, S. G. Design for parallel interactive ray tracing systems. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 187–196.
- [11] BOULOS, S., WALD, I., AND SHIRLEY, P. Geometric and arithmetic culling methods for entire ray packets. Tech. Rep. UUCS-06-10, School of Computing, University of Utah, August 2006.
- [12] BRIDSON, R., HOURIHAN, J., AND NORDENSTAM, M. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (SIGGRAPH '07)* 26, 3 (July 2007), 461–463.
- [13] BUCK, I. GPU computing with NVIDIA CUDA. In *GPGPU SIGGRAPH Course Notes* (August 2007).

- [14] CARPENTER, L. The A-buffer, an antialiased hidden surface method. *ACM SIGGRAPH Computer Graphics* 18, 3 (July 1984), 103–108.
- [15] CASSEN, T., SUBRAMANIAN, K. R., AND MICHALEWICZ, Z. Near-optimal construction of partitioning trees by evolutionary techniques. In *Graphics Interface '95* (May 1995), pp. 263–271.
- [16] CAUSTIC GRAPHICS. Introduction to CausticRT. In <http://www.caustic.com/> (July 2009).
- [17] CLARK, J. H. Hierarchical geometric models for visible surface algorithms. *ACM SIGGRAPH Computer Graphics* 10, 2 (July 1976), 267–267.
- [18] COOK, R. L., CARPENTER, L., AND CATMULL, E. The reyes image rendering architecture. *ACM SIGGRAPH Computer Graphics* 21, 4 (July 1987), 95–102.
- [19] COOK, R. L., AND DEROSE, T. Wavelet noise. *ACM Transactions on Graphics (SIGGRAPH '05)* 24, 3 (July 2005), 803–811.
- [20] COOK, R. L., PORTER, T., AND CARPENTER, L. Distributed ray tracing. *ACM SIGGRAPH Computer Graphics* 18, 3 (July 1984), 137–145.
- [21] DAMMERTZ, H., AND KELLER, A. Improving ray tracing precision by object space intersection computation. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 25–31.
- [22] DEMARLE, D. E., PARKER, S. G., HARTNER, M., GRIBBLE, C., AND HANSEN, C. Distributed interactive ray tracing for large volume visualization. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (October 2003), pp. 87–94.
- [23] DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. Razor: An architecture for dynamic multiresolution ray tracing. Tech. Rep. TR-07-52, Department of Computer Sciences, The University of Texas at Austin, January 2007.
- [24] DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. Faster ray tracing with SIMD shaft culling. Tech. Rep. MPI-I-2004-4-006, Max-Planck-Institut für Informatik, December 2004.
- [25] ERBER, T., AND HOCKNEY, G. M. Equilibrium configurations of N equal charges on a sphere. *Journal of Physics A: Mathematical and General* 24, 23 (December 1991), L1369–L1377.
- [26] FANG, H.-L., ROSS, P., AND CORNE, D. A promising genetic algorithm approach to job-shop scheduling, re-scheduling, and open-shop scheduling problems. In *Proceedings of the International Conference on Genetic Algorithms* (June 1993), pp. 375–382.

- [27] FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009* (August 2009), pp. 59–68.
- [28] FENDER, J., AND ROSE, J. A high-speed ray tracing engine built on a field-programmable system. In *Proceedings of the 2003 IEEE International Conference on Field-Programmable Technology* (December 2003), pp. 188–195.
- [29] FREE SOFTWARE FOUNDATION, INC. The GNU MP bignum library. In <http://gmplib.org/> (May 2009).
- [30] FUJIMOTO, A., TANAKA, T., AND IWATA, K. ARTS: Accelerated ray-tracing system. *Computer Graphics and Applications* 6, 4 (April 1986), 16–26.
- [31] GARANZHA, K. Efficient clustered BVH update algorithm for highly-dynamic models. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (August 2008), pp. 123–130.
- [32] GOLDSMITH, J., AND SALMON, J. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7, 5 (May 1987), 14–20.
- [33] GOLDSTEIN, R., AND NAGEL, R. 3-d visual simulation. *Simulation* 16, 25 (January 1971), 25–31.
- [34] GOVINDARAJU, V., DJEU, P., SANKARALINGAM, K., VERNON, M., AND MARK, W. R. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 2008 IEEE/ACM International Symposium on Microarchitecture* (November 2008), pp. 176–187.
- [35] GROSCH, H. R. J. Ray tracing with punched-card equipment. In *Proceedings of the Optical Society of America* (December 1945), vol. 35, p. 803A.
- [36] GROSCH, H. R. J. Ray tracing with the selective sequence electronic calculator. In *Proceedings of the Optical Society of America* (December 1949), vol. 39, p. 1059A.
- [37] GÜNTHER, J., FRIEDRICH, H., WALD, I., SEIDEL, H.-P., AND SLUSALLEK, P. Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum* 25, 3 (September 2006), 517–525.
- [38] GUSTAVSON, S. Simplex noise demystified. In <http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf> (March 2005).
- [39] HAINES, E. Bounding volume hierarchy formation: Two new ways. *Ray Tracing News* 21, 1 (November 2008).
- [40] HAMMING, R. W. *Digital Filters*, third ed. Dover Publications, Mineola, NY, June 1998.

- [41] HAVRAN, V. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, November 2000.
- [42] IZE, T., WALD, I., AND PARKER, S. G. Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization* (May 2007), pp. 101–108.
- [43] JONES, R. Intersecting a ray and a triangle with Plücker coordinates. *Ray Tracing News* 13, 1 (July 2000).
- [44] KAJIYA, J. T. The rendering equation. *ACM SIGGRAPH Computer Graphics* 20, 4 (August 1986), 143–150.
- [45] KAY, D. S., AND GREENBERG, D. Transparency for computer synthesized images. *ACM SIGGRAPH Computer Graphics* 13, 2 (August 1979), 158–164.
- [46] KAY, T. L., AND KAJIYA, J. T. Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics* 20, 4 (August 1986).
- [47] KENSLER, A. Tree rotations for improving bounding volume hierarchies. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (August 2008), pp. 73–76.
- [48] KENSLER, A., KNOLL, A., AND SHIRLEY, P. Better gradient noise. Tech. Rep. UUSCI-2008-001, SCI Institute, University of Utah, April 2008.
- [49] KENSLER, A., AND SHIRLEY, P. Optimizing ray-triangle intersection via automated search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 33–38.
- [50] KIRK, D., AND ARVO, J. Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*. Academic Press, San Diego, CA, October 1991, ch. V.4, pp. 264–266.
- [51] KIRKPATRICK, S., GELATT, JR., C. D., AND VECCHI, M. P. Optimization by simulated annealing. *Science* 220, 4598 (May 1983), 671–680.
- [52] KOPTA, D., SPUJT, J., BRUNVAND, E., AND PARKER, S. Comparing incoherent ray performance of TRaX vs. Manta. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing* (August 2008), p. 183. Poster.
- [53] LAFORTUNE, E. P., AND WILLEMS, Y. D. Bi-directional path tracing. In *Proceedings of the Third International Conference on Computational Graphics and Visualization Techniques* (December 1993), pp. 145–153.
- [54] LAGAE, A., LEFEBVRE, S., DRETTAKIS, G., AND DUTRÉ, P. Procedural noise using sparse Gabor convolution. *ACM Transactions on Graphics (SIGGRAPH '09)* 28, 3 (August 2009).

- [55] LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (September 2006), pp. 39–45.
- [56] LEWIS, J. P. Algorithms for solid noise synthesis. *ACM SIGGRAPH Computer Graphics* 23, 3 (July 1989), 263–270.
- [57] LEXT, J., ASSARSSON, U., AND MÖLLER, T. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications* 21, 2 (March 2001), 22–31.
- [58] LI, X., GARZARAN, M. J., AND PADUA, D. Optimizing sorting with genetic algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization* (March 2005), pp. 99–110.
- [59] LÖFSTEDT, M., AND AKENINE-MÖLLER, T. An evaluation framework for ray-triangle intersection algorithms. *Journal of Graphics Tools* 10, 2 (March 2005), 13–26.
- [60] MACDONALD, D. J., AND BOOTH, K. S. Heuristics for ray tracing using space subdivision. *Visual Computing* 6, 3 (1990), 153–166.
- [61] MAHOVSKY, J. A. *Ray Tracing With Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary, August 2005.
- [62] METROPOLIS, N., ROSENBLUTH, A. W., ROSENBLUTH, M. N., TELLER, A. H., AND TELLER, E. Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 21, 6 (June 1953), 1087–1092.
- [63] MÖLLER, T., AND TRUMBORE, B. Fast, minimum storage ray triangle intersection. *Journal of Graphics Tools* 2, 1 (October 1997), 21–28.
- [64] MORLEY, R. K., BOULOS, S., JOHNSON, J., EDWARDS, D., SHIRLEY, P., ASHIKHMIN, M., AND PREMOŽE, S. Image synthesis using adjoint photons. In *Proceedings of Graphics Interface 2006* (June 2006), pp. 179–186.
- [65] MÜLLER, G., AND FELLNER, D. W. Hybrid scene structuring with application to ray tracing. In *Proceedings of the 1999 International Conference on Visual Computing* (February 1999), pp. 19–26.
- [66] MUSGRAVE, F. K. Fractal solid textures: Some examples. In *Texturing and Modeling: A Procedural Approach*, third ed. Morgan Kaufmann Publishers, San Francisco, CA, December 2002, ch. 15, pp. 447–487.
- [67] MUUSS, M. J. Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium* (June 1995).
- [68] NG, K., AND TRIFONOV, B. Automatic bounding volume hierarchy generation using stochastic search methods. In *CPSC532D Mini-Workshop “Stochastic Search Algorithms”* (April 2003), pp. 147–161.

- [69] OLANO, M. Modified noise for evaluation on graphics hardware. In *Proceedings of Graphics Hardware 2005* (July 2005), pp. 105–110.
- [70] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP application program interface, v3.0. In <http://openmp.org/> (May 2008).
- [71] O’ROURKE, J. *Computational Geometry In C*, second ed. Cambridge University Press, New York, NY, September 1998.
- [72] PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. Interactive ray tracing. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics* (1999), pp. 119–126.
- [73] PARKER, S. G., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. Interactive ray tracing for isosurface rendering. In *Proceedings of the conference on Visualization '98* (October 1998), pp. 233–238.
- [74] PEACHEY, D. Building procedural textures. In *Texturing and Modeling: A Procedural Approach*, third ed. Morgan Kaufmann Publishers, San Francisco, CA, December 2002, ch. 2, pp. 7–94.
- [75] PERLIN, K. An image synthesizer. *ACM SIGGRAPH Computer Graphics* 19, 3 (July 1985), 287–296.
- [76] PERLIN, K. In the beginning: The pixel stream editor. In *Real-Time Shading SIGGRAPH Course Notes*, M. Olano, Ed. August 2001, ch. 2.
- [77] PERLIN, K. Noise hardware. In *Real-Time Shading SIGGRAPH Course Notes*, M. Olano, Ed. August 2001, ch. 9.
- [78] PERLIN, K. Improving noise. *ACM Transactions on Graphics (SIGGRAPH '02)* 21, 3 (July 2002), 681–682.
- [79] PERLIN, K. Implementing improved perlin noise. In *GPU Gems*. Addison-Wesley, Upper Saddle River, NJ, April 2004, ch. 5, pp. 73–85.
- [80] PERLIN, K., AND HOFFERT, E. Hypertexture. *ACM SIGGRAPH Computer Graphics* 23, 3 (July 1989), 253–262.
- [81] PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (SIGGRAPH '02)* 21, 3 (July 2002), 703–712.
- [82] RESHETOV, A. Faster ray packets – triangle intersection through vertex culling. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (September 2007), pp. 105–112.
- [83] RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. Multi-level ray tracing algorithm. *ACM Transactions on Graphics (SIGGRAPH '05)* 24, 3 (July 2005), 1176–1185.

- [84] RUBIN, S. M., AND WHITTET, T. A 3-dimensional representation for fast rendering of complex scenes. *ACM SIGGRAPH Computer Graphics* 14, 3 (July 1980), 110–116.
- [85] SCHMITTLER, J., WALD, I., AND SLUSALLEK, P. SaarCOR – a hardware architecture for realtime ray-tracing. In *Proceedings of EUROGRAPHICS Workshop on Graphics Hardware* (September 2002).
- [86] SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of Graphics Hardware* (August 2004), pp. 95–106.
- [87] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics (SIGGRAPH '08)* 27, 3 (August 2008), 1–15.
- [88] SHIRLEY, P., AND MORLEY, R. K. *Realistic Ray Tracing*. A. K. Peters, Natick, MA, July 2003.
- [89] SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *Journal of the ACM* 32, 3 (July 1985), 652–686.
- [90] SPJUT, J., KENSLER, A., AND BRUNVAND, E. Hardware-accelerated gradient noise for graphics. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI* (May 2009), pp. 457–462.
- [91] SPJUT, J., KOPTA, D., BOULOS, S., KELLIS, S., AND BRUNVAND, E. TRaX: A multi-threaded architecture for real-time ray tracing. In *IEEE Symposium on Application Specific Processors* (June 2008), pp. 108–114.
- [92] ČERNÝ, V. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications* 45, 1 (January 1985), 41–51.
- [93] VEACH, E., AND GUIBAS, L. J. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (August 1997), pp. 65–76.
- [94] VOORHIES, D., AND KIRK, D. Ray-triangle intersection using binary recursive subdivision. In *Graphics Gems II*. Academic Press, San Diego, CA, October 1991, ch. V.3, pp. 257–263.
- [95] WALD, I. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.
- [96] WALD, I. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing* (September 2007), pp. 33–40.

- [97] WALD, I., BOULOS, S., AND SHIRLEY, P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics* 26, 1 (January 2007), 6:1–6:18.
- [98] WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics (SIGGRAPH '06)* 25, 3 (August 2006), 485–493.
- [99] WALD, I., AND SLUSALLEK, P. State-of-the-art in interactive ray-tracing. In *Eurographics State of the Art Reports* (September 2001), pp. 21–42.
- [100] WALD, I., SLUSALLEK, P., AND BENTHIN, C. Interactive distributed ray-tracing of highly complex models. In *Proceedings of the EUROGRAPHICS Workshop on Rendering* (June 2001), pp. 274–285.
- [101] WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (EUROGRAPHICS '01)* 20, 3 (September 2001), 153–164.
- [102] WEGHORST, H., HOOPER, G., AND GREENBERG, D. P. Improved computational methods for ray tracing. *ACM Transactions on Graphics* 3, 1 (January 1984), 52–69.
- [103] WHITTED, T. An improved illumination model for shaded display. *Communications of the ACM* 23, 6 (June 1980), 343–349.
- [104] WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools* 10, 1 (2005), 49–54.
- [105] WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (SIGGRAPH '05)* 24, 3 (July 2005).
- [106] YOON, S.-E., CURTIS, S., AND MANOCHA, D. Ray tracing dynamic scenes using selective restructuring. In *Proceedings of the EUROGRAPHICS Workshop on Rendering* (June 2007), pp. 73–84.