GROVE CITY COLLEGE

ESTABLISHED 1876 • PENNSYLVANIA

Integer Ray Tracing

Jared Heinly, Shawn Recker, Kevin Bensema, Jesse Porch, and Christiaan Gribble
Department of Computer Science
Grove City College

Technical report
GCC-CS-001-2009

Department of Computer Science
Grove City College
100 Campus Drive
Grove City, PA 16127

13 October 2009

Abstract

Despite nearly universal support for the IEEE 754 floating-point standard on modern general-purpose processors, a wide variety of more specialized processors do not provide hardware floating-point units and rely instead on integer-only pipelines. Ray tracing on these platforms thus requires an integer rendering process. Toward this end, we clarify the details of an existing fixed-point ray/triangle intersection method, provide an annotated implementation of that method in C++, introduce two refinements that lead to greater flexibility and improved accuracy, and highlight the issues necessary to implement common material models in an integer-only context. Finally, we provide the source code for a template-based integer/floating-point ray tracer to serve as a testbed for additional experimentation with integer ray tracing methods.

Integer Ray Tracing

Jared Heinly Shawn Recker Kevin Bensema Jesse Porch Christiaan Gribble
Department of Computer Science
Grove City College

conference **kalabsha** **poolhall** **rtrt** **sponza**

Figure 1: *Integer ray tracing. A wide variety of specialized processors do not provide hardware floating-point units and rely instead on integer-only computational pipelines. We explore the problem of ray tracing on these platforms and describe a complete, integer-only ray tracing pipeline, including traversal, intersection, and shading.*

Abstract

Despite nearly universal support for the IEEE 754 floating-point standard on modern general-purpose processors, a wide variety of more specialized processors do not provide hardware floating-point units and rely instead on integer-only pipelines. Ray tracing on these platforms thus requires an integer rendering process. Toward this end, we clarify the details of an existing fixed-point ray/triangle intersection method, provide an annotated implementation of that method in C++, introduce two refinements that lead to greater flexibility and improved accuracy, and highlight the issues necessary to implement common material models in an integer-only context. Finally, we provide the source code for a template-based integer/floating-point ray tracer to serve as a testbed for additional experimentation with integer ray tracing methods.

1 Introduction

Whereas hardware support for the IEEE 754 floating-point standard [IEEE 2008] is common among general-purpose CPUs, many processors—such as low-power embedded processors, high-performance stream processors, specialized digital signal processors, and so forth—do not provide hardware support for IEEE 754. Leveraging these platforms for ray tracing thus requires an integer-only rendering process. Moreover, eliminating the need for floating-point units in the design of a custom ray tracing architecture can significantly reduce power and area requirements, but still deliver performance competitive with processors that provide full hardware support for the standard. In this context, we present a complete integer ray tracing pipeline targeting computational environments that lack hardware support for the IEEE 754 floating-point standard.

Although it is tempting to dismiss the problem of integer ray tracing as a matter of abstraction—that is, one solved simply by designing an appropriate fixed-point data type for use in place of standard floating-point types—our experience shows that a correct, efficient, and accurate solution involves a number of subtle details that complicate this view.

Fixed-point ray/triangle intersection. Recently, Hanika et al. [Hanika and Keller 2007; Hanika 2007] have explored fixed-point ray/triangle intersection, demonstrating that this core operation can be implemented without support for floating-point types. The method is ideal for a direct hardware implementation because eliminating floating-point hardware units reduces the power and area requirements of the architecture. We leverage Hanika's method in the integer-only ray tracing pipeline described here.

Contributions. To help others interested in the problem of integer ray tracing, we:

- highlight the caveats that arise in the implementation of an integer ray tracer, paying particular attention to the details necessary to implement the full rendering pipeline;
- describe a method of computing Hanika's triangle edge shift adaptively, avoiding hard-coded values to accommodate a wider range of scenes;
- carefully analyze ray/bounding box intersection to reveal unused bits and improve accuracy;
- clarify the details of Hanika's fixed-point ray/triangle intersection method with an annotated implementation in the C++ programming language; and
- outline the implementation of common material models in an integer-only context.

We also provide the source code for a template-based integer/floating-point renderer that can serve as a testbed for further enhancements and additional experimentation with integer ray tracing techniques. The renderer implements the complete ray tracing pipeline, including traversal, intersection, and shading. All of the components have been carefully designed to execute in computational environments that lack hardware support for IEEE 754 floating-point types.

2 Implementation

Integer ray tracing necessitates careful attention to details typically handled by IEEE 754 floating-point implementations.

2.1 Numeric spaces

The positions, vectors, and colors used throughout the rendering process are represented in distinct numerical domains that balance range and precision with respect to the values encoded in that space.

We follow the outline provided by Hanika: each component of a position in three-dimensional space is represented using a 29-bit

scene element	domain	# of bits [radix]	range	precision
Vertices	Integer	29 [0]	$[0, 2^{29}]$	1
Vectors	Fixed-point	32 [31]	$[-1, 1)$	2^{-31}
Colors	Fixed-point	32 [16]	$[0, 2^{16}]$	2^{-16}

Table 1: *Numeric spaces. The key values used in ray tracing are represented in distinct numeric spaces. The interaction between elements expressed in different domains requires careful attention in order to maintain a high degree of precision.*

unsigned integer, whereas vector components are encoded in signed 32-bit integers with the radix point to the right of bit 31. We also implement integer-only shading, so RGB color components are represented as unsigned 32-bit integers with the radix point to the right of bit 16. The key characteristics of these domains are summarized in Table 1.

These spaces are designed to balance the range/precision tradeoffs inherent to an integral approximation of real numbers, tradeoffs that are typically managed by the IEEE 754 implementation. In that representation, precision is traded for magnitude, and so IEEE 754 suffers from an unequal distribution of values along the real number line. In contrast, fixed-point representations are evenly distributed; thus, if the precision necessary to accurately represent a value is available near the origin, that precision is available anywhere in the space—fixed-point precision is invariant under translation.

Positions. Positions in three-dimensional space, including geometry vertices, bounding box extents, ray origins, camera and light source positions, and *t*-values in the ray parameter space are represented in a purely integer domain. As shown by Hanika, representing the length of the maximum displacement in this space with an unsigned 32-bit integer requires limiting the range of values to $[0, 2^{29}]$. Once appropriate shift and scale factors have been determined, scene elements represented in this space are mapped to the resulting integer grid, effectively discretizing the input domain.

Vectors. Vector quantities are expressed in a fixed-point representation with the radix point to the right of bit 31; each component of a vector quantity is thus in the range $[-1, 1)$ with 31 bits of precision:

$$\begin{aligned} 0.000000...00 &\rightarrow 0 \\ 0.000000...01 &\rightarrow 0 + 2^{-31} \\ &\dots \\ 0.111111...11 &\rightarrow 0 + \textstyle\sum_{i=1}^{31} 2^{-i} \end{aligned}$$

$$\begin{aligned} 1.000000...00 &\rightarrow -1 \\ 1.000000...01 &\rightarrow -1 + 2^{-31} \\ &\dots \\ 1.111111...11 &\rightarrow -1 + \textstyle\sum_{i=1}^{31} 2^{-i} \end{aligned}$$

In this domain, range is traded for precision; however, all vector components can be moved into the representable range by carefully managing values as they are manipulated throughout the rendering pipeline.

Colors. In addition to these vertex and vector spaces, we represent RGB color values as unsigned 32-bit integers in a fixed-point space with the radix point to the right of bit 16. In our experience, this representation captures color components in the range $[0, 1]$ reasonably well, while still permitting intensities greater than one for scenes utilizing physically accurate measures of light intensity.

Although mathematical operations with fixed-point values are well-defined, maintaining the highest degree of precision requires careful attention to the use of such operators. In addition, the interaction between computational elements expressed in the purely integer domain and those in each of the fixed-point representations complicate the programmer's ability to use the latter as a first class data type blindly throughout the rendering process.

```
typedef struct
{
  int np;  // Two shorter components of normal vector,
  int nq;  // each scaled by Nr in fixed-point space

  int r  :  2; // Index of longest component of normal
  int pp : 30; // p-component of v0 in vertex space
  int pq;      // q-component of v0 in vertex space

  int d;  // Distance to origin along scaled normal
          // in vertex space

  union
  {
    // Unbiased edge components in floating-point space
    struct { float e1pf, e1qf, e2pf, e2qf; };

    // Biased edge components in fixed-point space
    struct { int e1pi, e1qi, e2pi, e2qi; };
  };
} HanikaTriangle;
```

Figure 2: *Triangle representation. This data structure is used to convert a floating-point triangle to an integer representation.*

2.2 Scene discretization

In order to maintain compatibility with preexisting scenes, we implement a conversion process mapping floating-point scene elements to their integral counterparts. All positions in three-dimensional space are discretized by shifting vertices to the positive octant and by scaling according to a scene-dependent factor that maps the maximum component of the scene bounds in the floating-point domain to 2^{29} in the integer domain.

Triangle representation. Hanika's fixed-point ray/triangle intersection utilizes a refactored form of Wald's projection method [Wald 2004]. Triangles must be converted from the floating-point representation to the integral form prior to ray tracing. Figure 2 shows the intermediate data structure used during the triangle conversion process; we follow Hanika's derivation of these values.

Note that, during discretization, some triangles will be degenerate, exhibiting either zero-length edges or zero area. We apply two simple tests to detect and remove these triangles from the scene database during discretization, typically with little effect on the resulting image quality.

However, before moving edge components to the fixed-point domain, the values are biased according to an edge shift designed to reduce discretization artifacts. In particular, Hanika's analysis of triangle edge statistics across a number of test scenes shows that clamping values to the range $(-2^{-E}, 2^{-E})$ before discretization reduces the artifacts. According to this analysis, the edge shift $E = 10$ suffices to render the scenes tested. In our experience, however, discretization errors are visible with $E = 10$ for a different collection of scenes (see Section 3). In fact, these scenes require a much wider range ($E \in [3, 12]$), demonstrating that a single, hard-coded value may hamper the correct rendering of some scenes.

Refinement: scene-dependent edge shift. To manage this issue, we instead determine the edge shift adaptively during conversion. In particular, we track the maximum ratio of triangle edge

components to the maximum component of the face normal, N_r, across all triangles during the discretization process. This value is then used to compute the scene-dependent edge shift and bias:

```
0  //////////////////////////////////////
1  // Compute scene-dependent edge shift
2
3  int   E    = int(-ceilf(logf(max)/logf(2.f)));
4  float bias = powf(2.f, static_cast<float>(E));
```

The edges of each partially-converted triangle are then scaled accordingly:

```
0  ///////////////////////////////////
1  // Bias edges of current triangle
2
3  e1pi = flt2fxd(e1pf*bias);
4  e2pi = flt2fxd(e2pf*bias);
5  e1qi = flt2fxd(e1qf*bias);
6  e2qi = flt2fxd(e2qf*bias);
```

The biased edge components are now in the range $[-1, 1]$, which very nearly maps directly to the 31-bit fixed-point domain. However, rather than risk overflow for edge lengths equal to one (a common scenario in our experience), we scale all edges to the range $(-1, 1)$ during the fixed-point conversion (lines 3-6). Doing so introduces some additional error, but this error is bounded by the edge length times 2^{-31} in the floating-point domain.

Finally, the intermediate data can be copied to an integer-only data structure for use during rendering, complete with material identifiers and any other necessary information.

2.3 BVH Traversal

Ray/bounding box intersection for BVH traversal requires up to six division operations, which can become quite expensive, particularly in integer arithmetic. In floating-point arithmetic, a faster alternative instead multiplies by a reciprocal direction vector,

$$inv = (\frac{1}{d_x}, \frac{1}{d_y}, \frac{1}{d_z})$$

stored with each ray [Williams et al. 2005]. This method leads to significant performance improvements, and the storage cost of an additional vector per ray is amortized across the large number of bounding box intersections performed with each ray.

Fixed-point reciprocal directions. However, calculating a reciprocal direction in the fixed-point domain is not as straightforward as the floating-point equivalent. Precision suffers with the naive approach in which one is simply converted to its fixed-point representation and divided by each component.

In particular, observe that for any unit length vector, at least one component must be greater than or equal to $\frac{1}{\sqrt{3}} = 0.5774$. When the corresponding component of the reciprocal direction is computed, the result will always evaluate to ± 1 unit due to integer division: in fact, any component greater than $\frac{1}{2}$ will produce the same reciprocal value, and each unit length ray direction will contain at least one such component.

To overcome this issue, Hanika uses a 64-bit representation of the reciprocal direction and incorporates an additional left shift in the numerator, according to a parameter C, to effectively divide a value greater than one by the ray direction:

```
0  ///////////////////////////////////
1  // Compute reciprocal direction
2
3  dir[0] = (dir[0] == 0 ? 1 : dir[0]);
4  dir[1] = (dir[1] == 0 ? 1 : dir[1]);
5  dir[2] = (dir[2] == 0 ? 1 : dir[2]);
6
7  sign[0] = (dir[0] < 0);
8  sign[1] = (dir[1] < 0);
9  sign[2] = (dir[2] < 0);
10
11 int64 n = int64(1) << (31 + C);
12
13 inv[0] = (dir[0] ? n/dir[0] : (-2*sign[0] + 1)*n);
14 inv[1] = (dir[1] ? n/dir[1] : (-2*sign[1] + 1)*n);
15 inv[2] = (dir[2] ? n/dir[2] : (-2*sign[2] + 1)*n);
```

Here, if a component of the ray direction is zero, the corresponding value in the reciprocal is initialized to one with the appropriate sign (lines 13-15).

Note that artificially shifting the numerator in the reciprocal direction calculation (line 11) requires an equal and opposite shift later to yield correct results. In particular, the Williams-style ray/bounding box intersection now proceeds as follows:

```
0  ////////////////////////////////////////////////
1  // Williams-style ray/bounding box intersection
2
3  int64 tminX = (int64(bounds[  sign[0]][0] - org[0])
4    >> C)*inv[0];
5  int64 tmaxY = (int64(bounds[1-sign[1]][1] - org[1])
6    >> C)*inv[1];
7  if (tminX > tmaxY)
8    return false;
9
10 int64 tmaxX = (int64(bounds[1-sign[0]][0] - org[0])
11   >> C)*inv[0];
12 int64 tminY = (int64(bounds[  sign[1]][1] - org[1])
13   >> C)*inv[1];
14 if (tmaxX < tminY)
15   return false;
16
17 int64 tmin  = (tminX > tminY ? tminX : tminY);
18 int64 tmaxZ = (int64(bounds[1-sign[2]][2] - org[2])
19   >> C)*inv[2];
20 if (tmin > tmaxZ)
21   return false;
22
23 int64 tmax  = (tmaxX < tmaxY ? tmaxX : tmaxY);
24 int64 tminZ = (int64(bounds[  sign[2]][2] - org[2])
25   >> C)*inv[2];
26 if (tmax < tminZ)
27   return false;
28
29 tmin = (tmin > tminZ ? tmin : tminZ);
30 tmax = (tmax < tmaxZ ? tmax : tmaxZ);
31 if (tmin <= tmax)
32 {
33   if (tmin > INT_MAX || tmin < -INT_MAX)
34     return false;
35
36   t = int(tmin);
37   return true;
38 }
39
40 return false;
```

Note that the ray may intersect the plane of a box's side beyond the bounds of the scene, so in the integer domain, the resulting t-value may require 64 bits to encode correctly; promoting values to a 64-bit representation accounts for this subtlety (lines 3-6, 10-13, 18-19, and 24-25). The 64-bit result is first checked against the valid in-bounds interval (lines 33-34) and is then converted back to 32 bits before reporting a valid intersection (lines 36-37).

An obvious tradeoff exists in this intersection calculation: for each bit of precision given to the reciprocal direction, a bit of precision is

removed from the difference between the bounds and the ray origin (for example, lines 7-8). If the value of C is too low, insufficient precision exists to adequately represent the reciprocal direction and inaccuracies result. On the other hand, if the value is too high, the difference between the bound and the origin becomes inaccurate and bounding boxes are intersected erroneously. A balance must be struck between these two extremes in order to achieve acceptable results. In our experience, $C = 12$ balances the tradeoffs well: although inaccuracies may still arise, they are greatly reduced compared to the naive computation of the reciprocal direction.

Refinement: improving accuracy. Close inspection of ray/bounding box intersection reveals a simple improvement yielding greater accuracy. Consider the basic operation:

```
((bound - origin) >> C) * reciprocal
```

Here, both `bound` and `origin` are positions in the purely integer domain and are thus represented in 29 bits. The difference, however, may be negative, so 30 bits are actually required to represent this value correctly. The quantity is then shifted right by C, so the number of bits used by this component is actually $(30 - C)$.

Observe that the reciprocal direction is a vector in the fixed-point domain that has been shifted left by C bits. After the multiplication, a total of $(30 - C) + (32 + C) = 62$ bits are necessary to encode the result, leaving 2 bits in the 64-bit representation unused.

Further unused bits can be identified if we recognize that components of the reciprocal direction will very rarely require all $(32+C)$ bits; fewer bits will be required unless the component of the ray direction is (very nearly) equal to ± 1 unit. Therefore, by guaranteeing that the direction vector used in the reciprocal calculation has a magnitude greater than or equal to $2 = 2^1$, we can recover another unused bit. We can continue this process to recover even more unused bits: for example, if we know that the magnitude is greater than $4 = 2^2$, we can recover two bits; greater than $8 = 2^3$, three bits, and so forth.

To ensure the necessary constraints, we modify the reciprocal direction calculation to recover the desired number of bits: if the magnitude is less than the chosen value ($16 = 2^4$ in our case), the direction is temporarily scaled to the proper signed magnitude:

```
 0  ////////////////////////////////////////
 1  // Compute reciprocal direction (refined)

    <lines 2-12 omitted>

13  inv[0] = (abs(dir[0]) >= (1 << 4) ? n/dir[0] :
14            (-2*s[0] + 1)*n)) >> 4);
15  inv[1] = (abs(dir[1]) >= (1 << 4) ? n/dir[1] :
16            (-2*s[1] + 1)*n)) >> 4);
15  inv[2] = (abs(dir[2]) >= (1 << 4) ? n/dir[2] :
17            (-2*s[2] + 1)*n)) >> 4);
```

When these unused bits are combined with those discovered above, we recover a total of six unused bits that are used to improve the accuracy of the calculations: instead of shifting by C, values are shifted by a parameter D, the difference between C and the number of unused bits, lending additional precision to the result and reducing the number of visible artifacts relative to a floating-point implementation. We have empirically determined that $C = 15$ and $D = 9$ is optimal with respect to the artifacts visible in our examples.

This optimization increases accuracy but imposes a slight performance penalty: the t-value is effectively shifted left by $C - D$ bits. If the ray encounters a valid intersection, the t-value must first be shifted right by this difference. In our experience, the extra shift operation required with every successful bounding box intersection degrades performance by less than 2%.

2.4 Ray/triangle intersection

As noted, Hanika utilizes a refactored form of Wald's projection method designed specifically for integer ray tracing. Here, we annotate a C++ implementation of Hanika's method to highlight the caveats imposed by fixed-point arithmetic.

To maintain maximum precision and avoid overflow, the origin and direction of the ray are temporarily promoted to 64-bit values at the beginning of the intersection routine. We also determine indices of the p- and q-axes for the given triangle using a simple mod-3 lookup table:

```
 0  ////////////////////////////
 1  // Promote values to 64 bits
 2
 3  const Point& org       = ray.org();
 4  const int64  origin[3] = { org[0], org[1], org[2] };
 5
 6  const Vector& dir      = ray.dir();
 7  const int64   omega[3] = { dir[0], dir[1], dir[2] };
 8
 9  const uint p = mod3[r+1];
10  const uint q = mod3[r+2];
```

We first require the intersection of the ray and the plane of the triangle, with the ray direction projected onto the triangle normal:

```
11  ////////////////////////////
12  // Ray/plane intersection
13
14  int64 denom = omega[r] + ((omega[p]*np) >> 31) +
15                           ((omega[q]*nq) >> 31);
16  if (abs(denom) < epsilon)
17    return false;
18
19  int64 numer = origin[r] + ((origin[p]*np) >> 31) +
20                            ((origin[q]*nq) >> 31);
21  int64 tval  = numer / denom;
22  if (tval > INT_MAX || tval < -INT_MAX)
23    return false;
```

Note that shifting right by 31 bits (lines 14-15 and 19-20) brings the temporary 64-bit results back into the fixed-point domain.

If the intersection occurs within the valid interval, the p and q components of the hit point relative to the vertex v_0 are determined:

```
24  ////////////////////////////////////////
25  // Compute p, q components of hit point
26
27  int64 kp = origin[p] + ((tval*omega[p]) >> 31) - pp;
28  int64 kq = origin[q] + ((tval*omega[q]) >> 31) - pq;
```

Here, too, the right shift brings the temporary 64-bit results back into the fixed-point range.

Now, the barycentric coordinates of the hit point are computed:

```
29  ////////////////////////////////////
30  // Compute barycentric coordinates
31
32  int64 u = int64(e1p)*kq - int64(e1q)*kp;
33  if (u < 0)
34    return false;
35
36  int64 v = int64(e2q)*kp - int64(e2p)*kq;
37  if (v < 0)
38    return false;
39
40  if (((u + v) >> E) > (int64(1) << 31))
41    return false;
```

Recall that, in the integer representation, the triangle edge components are biased: a right shift by the edge shift E (line 40) is necessary to compensate for the bias applied during preprocessing.

Finally, if the barycentric coordinates are required for shading, `u` and `v` can be converted to the fixed-point space and packaged for return to the caller before reporting a valid intersection:

```
42  ////////////////////////////////////////////
43  // Prepare barycentric coordinates for return
44
45  beta  = (u >> E);
46  gamma = (v >> E);
47  alpha = (INT_MAX - beta - gamma);
```

This implementation is suitable for any platform with a standard C++ compiler and support for 64-bit integral data types.

2.5 Shading

In addition to traversal and intersection, we implement an integer-only shading process: ideal Lambertian, Blinn-Phong metal, and dielectric material models are supported by the Whitted-style recursive ray tracer provided online.

In general, integer-only shading follows its floating-point counterpart closely, with a few exceptions necessary to avoid overflow and maintain precision. We thus utilize a `FixedPoint` data type to simplify the implementation and improve readability:

```
0  ////////////////////////////////////////////
1  // Type definitions for a general fixed-point
2  // representation
3
4  template<int N>
5  class FixedPoint;
6
7  typedef FixedPoint<31> fp31;
8  typedef FixedPoint<16> fp16;
```

This class template overloads the basic arithmetic operations for a fixed-point representation with the radix point to the right of bit `N`, and is used throughout shading to shield the programmer from the details of the fixed-point representation wherever possible.

Reflection direction. The computation of the mirror reflection direction, $V - 2(N \cdot V)N$, necessitates scaling the ray direction and the shading normal to avoid overflow:

```
0  ////////////////////////////////////////
1  // Compute mirror reflection direction
2
3  fp31 cosTheta = dot(n, dir);
4
5  rdir[0] = ((dir[0] >> 2) - 2*(cosTheta*(n[0] >> 2)))
6    << 2;
7  rdir[1] = ((dir[1] >> 2) - 2*(cosTheta*(n[1] >> 2)))
8    << 2;
9  rdir[2] = ((dir[2] >> 2) - 2*(cosTheta*(n[2] >> 2)))
10   << 2;
```

Note that the computation proceeds component-wise because scaling $N \cdot V$ by two involves a value falling outside the range of the 31-bit fixed-point domain. Overloaded operators implement the operations necessary to compute the mixed-type product $2(N \cdot V)N$, including the 64-bit promotions and subsequent shifts to move the results back into the 31-bit fixed-point space.

Schlick's approximation. We use Schlick's approximation [Schlick 1993; Schlick 1994] to compute the Fresnel terms in specular reflection, represented in the 31-bit fixed-point domain:

```
0  ////////////////////////////////////
1  // Compute Schlick's approximation
2
3  fp31 k  = fp31::One + cosTheta;
4  fp31 k2 = k*k;
5  fp31 k4 = k2*k2;
6  fp31 k5 = k4*k;
7
8  RGB R = R0 + (1 - R0)*(k5 >> 15);
```

Recall that RGB colors exist in a 16-bit fixed-point space, so the approximation must be shifted right 15 bits in order to move the value into the range of the 16-bit fixed-point representation (line 8).

Blinn-Phong reflection. We implement the Blinn-Phong model [Blinn 1977] for specular highlights. In this model, the halfway vector is computed by subtracting the ray direction from the direction to the light source, which requires scaling both vectors to avoid overflow:

```
0 ////////////////////////////////////
1 // Compute Blinn-Phong specular term
3
4  Vector H  = ((ldir >> 2) - (dir >> 2)).normal();
5  fp31   Hn = dot(H, n);
6  if (Hn > 0)
7  {
8    fp31 scale = pow(Hn, exp);
9    light += lcolor*(scale >> 15);
10 }
```

The term is calculated by raising $H \cdot N$ to a power specified by the object's material properties using an integer power function [Warren 2003]. The result is shifted right 15 bits to move the value into the 16-bit fixed-point color space, which then modulates the incident light to compute the specular contribution.

Index of refraction. Dielectric materials, such as glass, diamond, and so forth, are specified in part by an index of refraction, or the ratio of the material's optical density relative to vacuum. To properly account for light both entering and exiting dielectric objects, an implementation requires both this ratio and its reciprocal—values greater than one as well as less than one. To strike a balance, these values are encoded in the 16-bit fixed-point domain. For example, the relevant values for crown glass can be computed as follows:

```
double ETA    = 1.5;
fp16   eta    = fp16(ETA);
fp16   invEta = fp16(1./ETA);
```

Clearly this choice sacrifices some precision in the encoded values, but it greatly simplifies the dielectric shader implementation by abstracting the details of the fixed-point representation.

3 Examples

To demonstrate the integer ray tracing techniques described here, we render the scenes depicted in Figure 1. Table 2 summarizes the pertinent characteristics of these scenes, which represent a wide range of geometric complexity, particularly with regard to the relative scale of the triangles. In the *kalabsha* scene, for example, more than $\frac{1}{3}$ of the triangles are degenerate in the integer domain, and most scenes contain at least a few such triangles. The actual number of triangles rendered is thus some percentage of the total number of triangles in the original scene.

Performance. To characterize the impact of an integer-only environment on performance, we render each scene using the template-based integer/floating-point ray tracer available online.

Figure 3 shows the results when rendering the scenes on a 3.06 GHz Intel Core2 Duo processor according to the following rendering configuration:

scene	degen	triangles actual	% orig
conference	0	251064	100.0
kalabsha	732224	1391777	65.5
poolhall	2	83843	100.0
rtrt	2909	285805	99.0
sponza	20	76063	100.0

Table 2: *Test scenes. These scenes represent a wide range of geometric complexity, particularly with regard to the ratio between the minimum and maximum triangle edge lengths within the scene.*

parameter	value
image resolution	1024×1024
BVH leaf threshold	1
# of samples per pixel	1
# of shadow rays	1 per source
maximum ray depth	10
reciprocal dir: C bits only	$C = 12$
reciprocal dir: C/D bits	$C = 15, D = 9$

The refined reciprocal direction computation described above (D bits) results in a penalty of less than 2% relative to Hanika's original method (C bits). However, both versions of the integer ray tracer significantly underperform the floating-point renderer: performance differs by a factor of 2×–5× on the test machine, indicating that modern general-purpose CPUs are heavily optimized for floating-point operations. If performance is of concern, the use of integer ray tracing is not recommended on platforms that support IEEE 754 data types.

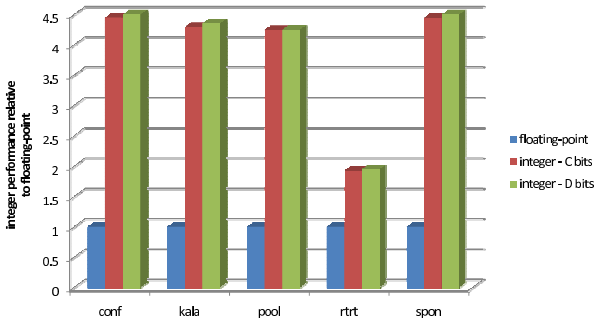
Image quality. To characterize the image quality realized with integer ray tracing, we conduct a simple comparison of the integer results to those produced by the floating-point renderer. In particular, ImageMagick [ImageMagick 2009] is used to generate difference images between the integer and floating-point results with the following command:

```
convert <float_image> <integer_image> \
        -compose difference \
        -composite \
        -separate \
        -background black \
        -compose plus \
        -flatten \
        <difference_image>
```

This command generates a gray scale image in which brighter pixels depict greater differences between the corresponding pixels. Therefore, regions with little difference appear black, whereas regions exhibiting some artifact are various shades of gray to white.

The top panel of Figure 4 illustrates the results with the *kalabsha* scene. Although some differences arise due to quantization errors in the integer shading process, we attribute the overwhelming majority of the visible differences to errors in ray/bounding box intersection: a dramatic reduction in the number of differences ensues if either the bounds of leaf nodes are artificially increased or the refinement for improved accuracy in ray/bounding box intersection is used.

As such, we examine the artifacts arising from the two reciprocal direction computation techniques more carefully. In particular, we use a simple counting strategy to quantify the visible differences: for each difference image, we count the number of pixels above a given threshold as a very rough approximation to more sophisticated, perceptually-based difference metrics [Daly 1993; Ramasubramanian et al. 1999].



scene	IEEE 754	integer C bits	v. 754	integer D bits	v. 754
conference	2.60	11.57	4.45×	11.75	4.52×
kalabsha	5.12	21.95	4.29×	22.27	4.35×
poolhall	5.97	25.32	4.24×	25.34	4.25×
rtrt	5.22	10.04	1.92×	10.18	1.95×
sponza	3.71	16.48	4.44×	16.73	4.51×

Figure 3: *Rendering time in seconds. Performance of the integer ray tracer relative to the floating-point renderer, both with and without the reciprocal direction refinement. Modern general-purpose CPUs are heavily optimized for floating-point performance: the integer renderer performs about 2×–5× slower than the floating-point renderer on the same hardware.*

The results are illustrated in the bottom panel of Figure 4. For these data, we use a threshold of 25%: if the difference between the integer and floating-point images is greater than 25% of the maximum—that is, an absolute difference of greater than 64 units in the 8-bit gray scale pixel values—the artifact is counted. Note that the refined computation reduces the number of differences in all scenes, in some cases quite significantly. We suggest that the modest impact on performance imposed by the D bits optimization is thus a favorable tradeoff.

4 Discussion

The primary utility of integer ray tracing is in environments without hardware support for IEEE 754 floating-point types. As the results in the previous section illustrate, modern general-purpose CPUs are heavily optimized for floating-point operations, so the use of integer ray tracing is not recommended for these platforms when performance is of concern.

Some other issues in integer ray tracing also deserve consideration:

- **Relative scale.** Scenes in which the geometry occupies a large spatial extent relative to the minimum triangle edge length can be problematic. An extreme example would be a scene that models everyday objects, lit by the sun positioned at a physically accurate distance. Capturing that scale will leave very few bits to represent the objects within the scene. The inability of integers to represent as vast a range of values as a floating-point representation limits the type of scenes that can be rendered with reasonable accuracy using integer ray tracing.
- **Epsilon.** With integer ray tracing, the error margin typically used to avoid the self-intersection problem for secondary rays [Woo et al. 1996] is no longer scene-dependent: all scenes are scaled into the same range and vertices are fixed to a raster grid. Hanika [Hanika 2007] shows that $\epsilon = 2$ suffices when shifting the origins of secondary rays along the

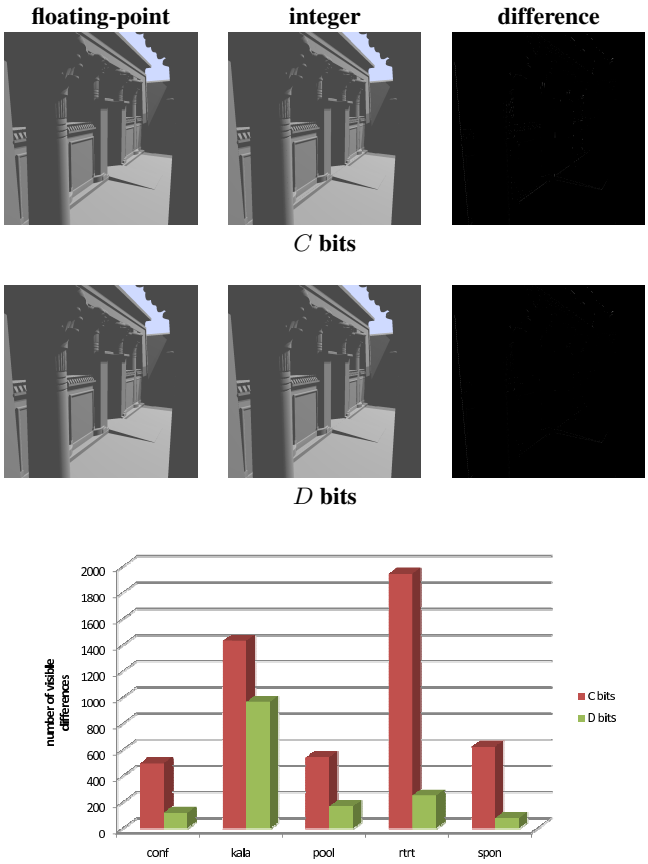


Figure 4: *Visible differences. Although some differences arise due to color quantization in the integer shading process, the overwhelming majority of the visible differences arise due to errors in ray/bounding box intersection. However, using the refinement for improved accuracy in ray/bounding box intersection significantly reduces the number of visible artifacts in the integer images.*

normal direction. However, we find $\epsilon = 6$ is required in our examples. The difference arises because of precision lost during shading. In particular, we often scale vectors to avoid overflow when computing the required values, and these bits cannot be recovered. Nevertheless, the value remains independent of the geometry and need not be adjusted from one scene to the next.

- **Seams.** When rendering images of the example scenes from various viewpoints, occasionally artifacts along the seam between two axis-aligned triangles appear. For example, when rendering the *conference* scene, seams along an edge of the door frame appear: rays cast toward that edge simply miss the geometry altogether. We attribute the issue to imprecision in ray/bounding box intersection: as before, if either the bounds of leaf nodes are artificially increased or the refinement for improved accuracy in ray/bounding box intersection is used, the visible differences are reduced significantly.

- **BVH depth.** Likewise, the leaf creation threshold used in the BVH construction process also impacts the relative image quality. For example, when using C bits with the *conference* scene, 497 differences are found when using a leaf creation threshold of one; when that threshold is increased to 64, only 111 of the differences remain. In general, we have found that increasing the BVH leaf threshold leads to fewer visible differences but increases rendering times. However, shallow BVHs can be beneficial in packet-based ray tracing techniques [Reshetov 2007; Dammertz et al. 2008], so this observation may be of value in this context.

We have explored the problem of integer ray tracing, specifically targeting platforms that lack hardware support for the IEEE 754 floating-point standard. It is our hope that these integer-only techniques will make ray tracing accessible to a wider range of useful, but non-traditional, processing environments.

Acknowledgments. This work was supported by a grant from the Swezey Scientific Instrumentation and Research Fund. The authors gratefully acknowledge the assistance of Johannes Hanika at Universität Ulm throughout the early stages of this project.

References

BLINN, J. F. 1977. Models of light reflection for computer synthesized pictures. *Computer Graphics 11*, 2 (July), 192–198.

DALY, S. 1993. The visible differences predictor: an algorithm for the assessment of image fidelity. In *Digital images and human vision*. MIT Press, 179–206.

DAMMERTZ, H., HANIKA, J., AND KELLER, A. 2008. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4, 1225–1233.

HANIKA, J., AND KELLER, A. 2007. Towards hardware ray tracing using fixed point arithmetic. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*, 119–128.

HANIKA, J. 2007. *Fixed Point Hardware Ray Tracing*. Master's thesis, Universität Ulm.

IEEE, 2008. IEEE standard for floating-point arithmetic, August.

IMAGEMAGICK, 2009. Imagemagick: Convert, edit, and compose images. Available at `http://www.imagemagick.org`.

RAMASUBRAMANIAN, M., PATTANAIK, S. N., AND GREENBERG, D. P. 1999. A perceptually based physical error metric for realistic image synthesis. In *Siggraph '99*, 73–82.

RESHETOV, A. 2007. Faster ray packets-triangle intersection through vertex culling. In *2007 IEEE Symposium on Interactive Ray Tracing*, 105–12.

SCHLICK, C. 1993. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, 73–84.

SCHLICK, C. 1994. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum 13*, 233–246.

WALD, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University.

WARREN, H. S. 2003. *Hacker's Delight*. Addison-Wesley, Upper Saddle River, New Jersey.

WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. 2005. An efficient and robust ray-box intersection algorithm. *Journal of Graphics, GPU, and Game Tools 10*, 1, 49–54.

WOO, A., PEARCE, A., AND OUELLETTE, M. 1996. It's really not a rendering but, you see... *IEEE Computer Graphics and Applications 16*, 5 (September), 21–25.