Robust BVH Ray Traversal - revised

Thiago Ize
Solid Angle

Figure 1. Left: Holes in the flower are ray-triangle intersections missed due to limited precision. Center: Many holes closed with our 1-ulp solution. Right: All holes closed with our 2-ulp solution.

Abstract

Most axis-aligned bounding-box (AABB) based BVH-construction algorithms are numerically robust; however, BVH ray traversal algorithms for ray tracing are still susceptible to numerical precision errors. We show where these errors come from and how they can be efficiently avoided during traversal of BVHs that use AABBs.

1. Introduction

BVHs are a popular acceleration structure for ray tracing, and almost all BVH-based ray tracers use axis-aligned bounding boxes as their bounding volumes since they are easy to build and fast to traverse. Therefore, we will focus solely on this type of BVH and the numerical precision errors that can occur when traversing them with a ray.

BVHs are built using numerically robust min/max operations over the underlying geometry, and so, the resulting tree will have no numerical precision issues, and a robust traversal algorithm will be able to correctly traverse the tree and find all valid primitives for ray-primitive intersection testing. Unfortunately, precision errors incurred during traversal can cause both false-hits and false-misses to occur. A false-hit occurs when a node is incorrectly visited even though the ray never actually entered that node. Aside from the performance penalties incurred by the unnecessary traversal,

this is a benign error since it will not result in any additional primitives being marked as hit. False-misses, on the other hand, occur when the traversal algorithm marks a ray as missing a node even though it does in fact enter that node. This error can lead to a valid ray-primitive intersection being missed if that primitive belonged to the missed node. During rendering, this could manifest as visually objectionable holes in a mesh, as seen in Figure 2, or in light leaking into a closed scene, which would introduce hard to identify speckles in Monte Carlo lighting simulations. A single false-miss could even introduce substantial overhead, such as when lazy loading is used. This can, for instance, happen if the ray escapes a closed room and hits a hundred million triangle object in the adjacent room, which must then be loaded from disk and have its BVH constructed. This paper concerns itself with removing false-misses due to the BVH traversal; of course, false-misses can occur from other places in a ray tracer, such as in the ray-primitive intersection, but those are orthogonal to this paper, since making one robust will not make the other robust.

Generally, false-hits and false-misses are rare, and so, from a performance perspective, these errors are normally not an issue, since they happen so infrequently. However, even just a single pixel with a hole can be visually objectionable. A naive attempt at solving this is to filter out the artifact with increased pixel samples, but this has a significant cost, and sometimes, increased sampling will not help at all, since the additional rays might also introduce error, which means that while some pixels will start to look better, other pixels that used to be correct with a single sample will in turn have error introduced by the new incorrect samples. Another issue with this approach is that the false-miss might cause something extremely bright, e.g., the sun, to be hit, which could require a prohibitive number of samples in order to filter out. Artifacts can be seen in Figure 2, despite 256 samples per pixel being cast. A somewhat better approach is to enlarge the bounding boxes of the BVH leaf nodes by a certain amount. However, as we will show later, this approach will always fail, since no matter what amount of padding is chosen, if the ray origin is moved sufficiently far from the bounding box, then false-misses can once again occur. In addition, this can result in a noticeable performance degradation if the padding is too large. Another attempted solution is to use a floating-point representation with more bits—for instance, going from single precision floats to double precision. This works fairly well by making the likely-hood of a false-miss much more rare, but a false-miss can still happen and this comes with a substantial performance and memory overhead, especially if SSE is being used for the BVH traversal, since that halves the throughput from four simultaneous float operations to two simultaneous double operations.

Mahovsky showed in his dissertation how a BVH traversal based on integer arithmetic could be made robust [Mahovsky 2005]. His approach was focused on ensuring traversal with low-precision integers would still be as robust and accurate as the higher quality floating-point traversal algorithms. However, it is not shown whether this

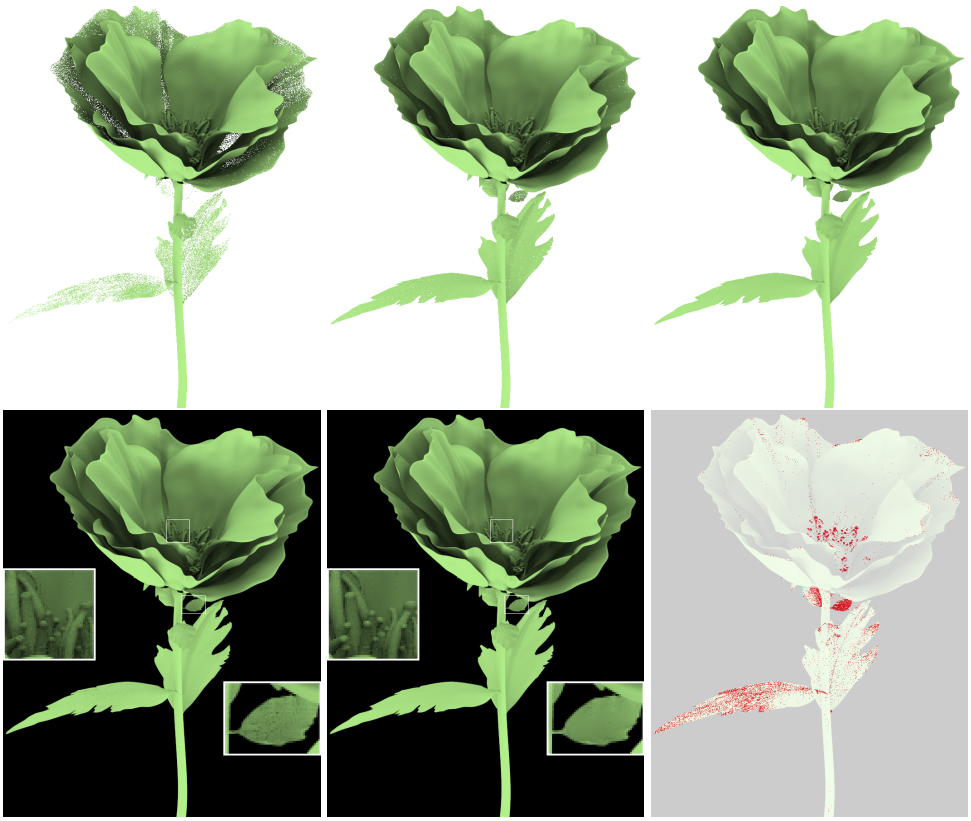

Figure 2. We rendered the 37.9M triangle flower at a 700×900 resolution with 256 samples per pixel using an ambient occlusion shader. The top row uses a very bright background so that even a single false-miss that strikes the background will still result in a completely white pixel. False-misses that end up hitting other parts of the flower tend to be averaged away which is why some parts of the flower appear not to have holes. From top left to top right, we rendered using no padding of any kind, one, and then two ulps with the robust InvUlps algorithm. Note that one ulp still results in false-misses while two ulps renders correctly without holes (the two small holes near the base of the petals are not precision issues but actual very tiny gaps between petals). The bottom row has a black background so that it is easier to filter away artifacts, with the left image using no padding, the middle image two ulps of padding, and the right image shows where these two image are different. In addition to the obvious holes, notice that the flower stamen are darker in the first image due to rays that accidentally entered into the unlit stamen.

approach is more accurate than his reference implementation using floats and it is possible that rounding errors in the floating-point computations used to produce the integers could result in a false-miss. If so, it probably would not be difficult to avoid this by appropriately expanding the relevant integers by one. In any case, while this could be implemented on a regular CPU, Mahovsky claimed it would be slower than a

regular floating-point traversal as it introduces extra complexity and certain expensive operations, such as bit shifts, which a hardware implementation could avoid. So, Mahovsky's integer arithmetic traversal is not generally applicable.

Since false-hits and false-misses normally are rare and do not affect render time, we do not bother removing false-hits and in fact trade the removal of false-misses for a negligible increase in false-hits. Our approach is to analyze where the error comes from, and from this, we discover two elegant and robust solutions. The first only requires adding a small amount of padding to our inverse ray direction used during BVH traversal without adding any extra computation during the BVH traversal loop. The second method shows that multiplying the max slab hit distance by a special constant float value will also be robust and extremely simple to implement, though it adds the slight overhead of an extra multiplication during each traversal step.

2. Current almost-robust traversal algorithm

We use the ray-bounding box test of Williams et al. for our BVH traversal [2005] combined with the simple improvement proposed by Berger-Perrin for avoiding NaNs [2004].

The Williams et al. test marks whether a ray intersects a bounding box by computing the ray-plane entrance and exit intersection distances for each of the three bounding box slabs and then returning a hit if the largest entering distance for the three slabs is smaller than the smallest exiting distance for the three slabs [Kay and Kajiya 1986]. The equation for computing the ray-plane intersection of the x-plane $b.x$ is $t = \frac{b.x - ray_{\text{origin}.x}}{ray_{\text{dir}.x}}$. The slabs in the y and z dimension are computed similarly. Williams et al. showed how this can be both optimized and made more robust in the case the ray direction component is zero, by precomputing the reciprocal of the direction so that the calculations performed, for the purposes of computing the numerical error, appears as $t = (b.x - ray_{\text{origin}.x})\frac{1}{ray_{\text{dir}.x}}$ [Williams et al. 2005].

Berger-Perrin made the traversal more robust for the case when the ray origin lies on the surface of the bounding box and the ray direction has a zero component, since in that case, a NaN is produced. Assuming $b.x = ray_{\text{origin}.x}$ and $ray_{\text{dir}.x} = 0$, we get

$$t = (b.x - ray_{\text{origin}.x})\frac{1}{ray_{\text{dir}.x}}$$
$$t = (0)\frac{1}{0} = 0\texttt{inf} = \texttt{NaN}$$

He handles this by carefully using standard min/max operations that are algorithmically defined as in Listing 1 so that if there is a NaN, the second argument will be returned, since a NaN equality always evaluates to false. Note that the SSE min/max assembly instructions, which is what Berger-Perrin uses, follows this, but the `std::min` / `std::max` do not always follow this rule and cannot be safely used. With this property,

```
template<class T>
const T& min(const T& a, const T& b) {
    return (a < b) ? a : b;
}

template<class T>
const T& max(const T& a, const T& b) {
    return (a > b) ? a : b;
}
```

Listing 1. NaN-safe min/max functions that will return NaN only if the second argument, `b`, is NaN

by making sure the argument that could have a NaN, such as t, is always placed as the first argument and the second argument never has a NaN, then no NaNs will be returned.

3. Sources of error

Given the current state-of-the-art robust BVH traversal, let us examine what error is still present. The bounding box planes given from b we know have no error, and the ray origin and direction are the same as used for the ray-primitive intersection test, so we assume they too have no error. This means the error can only be introduced from the subtraction, multiplication, and division operations and that catastrophic cancellation cannot occur when subtracting ray_{origin} from b, since they are perfectly represented in the input float.

IEEE 754 essentially dictates that each of these arithmetic operations is fully accurate, but the result must be rounded to the nearest representable floating point value (note that some processors, particularly older GPUs, are not IEEE 754 compliant—see for instance [Whitehead and Fit-Florea 2011]). The distance between a float x and the next closest float to it is defined as $ulp(x)$, where *ulp* stands for *unit in the last place*. This allows us to compute the maximum error introduced by an arithmetic operation as being the rounding error when the true result, given by the real number X, must be rounded to the floating-point x. For normal floating-point numbers, the magnitude of this error is at most

$$\frac{ulp(X)}{2} \leq |X|\varepsilon \tag{1}$$

where ε is the *machine epsilon*. For a 32 bit IEEE754 float, the mantissa has 24 bits (the first bit is implicit) so $\varepsilon = 2^{-24}$.

3.1. Traversal error

The ray-plane test will thus produce an approximate t, $\tilde{t}$, that is at most

$$\tilde{t} = \left(\left(\left(b - ray_{\text{origin}} \right) (1+\delta_1) \right) \left(\frac{1}{ray_{\text{dir}}} (1+\delta_2) \right) \right) (1+\delta_3) \tag{2}$$

$$\tilde{t} = (b - ray_{\text{origin}}) \frac{1}{ray_{\text{dir}}} (1+\delta_1)(1+\delta_2)(1+\delta_3) \tag{3}$$

$$\tilde{t} = (b - ray_{\text{origin}}) \frac{1}{ray_{\text{dir}}} (1+\delta_1+\delta_2+\delta_3+\delta_1\delta_2+\delta_1\delta_3+\delta_2\delta_3+\delta_1\delta_2\delta_3) \tag{4}$$

where the δ are the relative errors introduced by each floating point operation, and $|\delta| \leq \varepsilon$. Dropping the higher order δ terms, which are insignificant compared to floating-point precision, gives us

$$\tilde{t} = (b - ray_{\text{origin}}) \frac{1}{ray_{\text{dir}}} (1+\delta_1+\delta_2+\delta_3) \tag{5}$$

$$\tilde{t} = t(1+\delta_1+\delta_2+\delta_3) \tag{6}$$

The error in the floating point computed $\tilde{t}$ can be bounded by

$$|\tilde{t} - t| \leq |\delta_1 t| + |\delta_2 t| + |\delta_3 t| \tag{7}$$

$$|\tilde{t} - t| \leq 3\varepsilon |t| \tag{8}$$

$$|\tilde{t} - t| \leq 3\frac{ulp(t)}{2}, \tag{9}$$

and so the floating point computed $\tilde{t}$ is up to $\frac{3}{2}$ ulps away from the real distance t.

As shown in Figure 3, a false-miss can occur when the computed floating-point entry distance, $\tilde{t}_{min}$, is larger than the computed floating-point exit distance, $\tilde{t}_{max}$, even though the real exit distance is larger than or equal to the real entry distance. In other words, the following two conditions must occur for a false-miss:

$$\begin{aligned} &\tilde{t}_{min} > \tilde{t}_{max} \\ &t_{min} \leq t_{max}. \end{aligned} \tag{10}$$

Plugging in the $\frac{3}{2}$ ulps error bounds, the above condition can be simplified as

$$0 \leq t_{max} - t_{min} < 3ulp(t_{max}) \tag{11}$$

Since floats can only be an integer number of ulps away from each other, we get

$$0 \leq t_{max} - t_{min} \leq 2ulp(t_{max}) \tag{12}$$

Thus, false-misses can occur only if the entry and exit distances are within 2 ulps or less of each other.

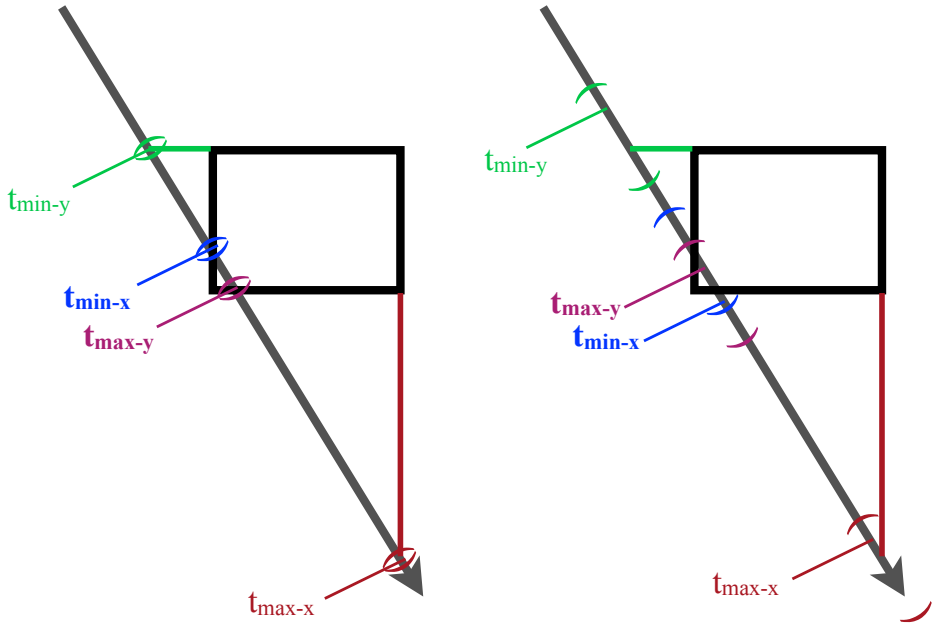

Figure 3. The ray is said to intersect the bounding box if the largest entry plane intersection distance, t_{min}, is smaller than or equal to the smallest exit plane distance, t_{max}. In the left example where there is no numerical error, this is the case, with $t_{min\text{-}x} < t_{max\text{-}y}$. However, in the right example we increase the precision error, denoted by the enlarged error bounds, and it is now possible for $\tilde{t}_{min}$ to become larger than $\tilde{t}_{max}$.

3.2. FMA optimized traversal error

If an IEEE754 compliant fused multiply-add (FMA) hardware instruction is available, then since the subtraction and multiplication operations become a single operation, only half an ulp of error is introduced when doing both a multiply and an add [Whitehead and Fit-Florea 2011]. In this case, in exchange for an additional multiplication at the start of the traversal, we can achieve faster traversal by using an FMA operation. We can see how to do this by taking our original equation and transforming it as such:

$$t = \left(b - ray_{\text{origin}}\right) \frac{1}{ray_{\text{dir}}} \tag{13}$$

$$t = b\frac{1}{ray_{\text{dir}}} + \left(-ray_{\text{origin}} \frac{1}{ray_{\text{dir}}}\right) \tag{14}$$

$$t = bY + X, \tag{15}$$

where X can be precomputed at the start of the BVH traversal as $-ray_{\text{origin}}\frac{1}{ray_{\text{dir}}}$ and Y as $\frac{1}{ray_{\text{dir}}}$. The final error, unfortunately, is not the same. The computed value is

$$\tilde{t} = \left(b\left(\frac{1}{ray_{\text{dir}}}(1+\delta_1)\right) + \left(-ray_{\text{origin}}\left(\frac{1}{ray_{\text{dir}}}(1+\delta_1)\right)\right)(1+\delta_2)\right)(1+\delta_3) \quad (16)$$

$$\tilde{t} = \frac{b}{ray_{\text{dir}}}(1+\delta_1)(1+\delta_3) - \frac{ray_{\text{origin}}}{ray_{\text{dir}}}(1+\delta_1)(1+\delta_2)(1+\delta_3) \quad (17)$$

$$\tilde{t} = (\frac{b}{ray_{\text{dir}}} - \frac{ray_{\text{origin}}}{ray_{\text{dir}}})(1+\delta_1)(1+\delta_3) - \frac{ray_{\text{origin}}}{ray_{\text{dir}}}(1+\delta_1)(1+\delta_3)\delta_2 \quad (18)$$

$$\tilde{t} = t(1+\delta_1)(1+\delta_3) - \frac{ray_{\text{origin}}}{ray_{\text{dir}}}(1+\delta_1)(1+\delta_3)\delta_2 \quad (19)$$

$$\tilde{t} = t(1+\delta_1+\delta_3+\delta_1\delta_3) - \frac{ray_{\text{origin}}}{ray_{\text{dir}}}(\delta_2+\delta_1\delta_2+\delta_2\delta_3+\delta_1\delta_2\delta_3) \quad (20)$$

where as before, $|\delta| < \varepsilon$ and give the relative errors introduced by each floating point operation. Dropping the higher order δ terms gives us

$$\tilde{t} = t + t(\delta_1+\delta_3) - \frac{ray_{\text{origin}}}{ray_{\text{dir}}}\delta_2. \quad (21)$$

The error is thus

$$\tilde{t} - t = t(\delta_1+\delta_3) - \frac{ray_{\text{origin}}}{ray_{\text{dir}}}\delta_2, \quad (22)$$

and can be bounded as

$$|\tilde{t} - t| \leq |t\delta_1| + |t\delta_3| + \left|\frac{ray_{\text{origin}}}{ray_{\text{dir}}}\delta_2\right| \quad (23)$$

$$|\tilde{t} - t| \leq 2\varepsilon|t| + \left|\frac{ray_{\text{origin}}}{ray_{\text{dir}}}\right|\varepsilon \quad (24)$$

The $\left|\frac{ray_{\text{origin}}}{ray_{\text{dir}}}\right|\varepsilon$ term can contribute significant amounts of error if it is larger than $ulp(t) = 2\varepsilon|t|$. A simple way to cause this to happen is to make $|t|$ small and $\left|\frac{ray_{\text{origin}}}{ray_{\text{dir}}}\right|$ large. This occurs when ray_{origin} and b are very close together, ray_{origin} has large magnitude, and ray_{dir} is small. Listing 2 gives an example of this where the non-FMA 32-bit floating point version has no ulps of error (at 32-bit precision), while the FMA version has 470,208 ulps of error. For this reason, the FMA optimization should be avoided when accuracy is important.

4. Robust traversal algorithm

We can make the non-FMA ray-box intersection completely robust by pushing the entry and exit plane intersection distances two ulps apart. One way to do this would be to use the `add_ulp_magnitude()` function from Listing 3 to modify the plane intersection inequality from $\tilde{t}_{min} < \tilde{t}_{max}$ to

$$\tilde{t}_{min} < \texttt{add\_ulp\_magnitude}(\tilde{t}_{max}, 2) \quad (25)$$

```
float b    = 1/6.99999;
float orig = 1/7.00000;
float dir  = 1e-6;
double b_d = b, orig_d = orig, dir_d = dir;

double t_ref = (b_d - orig_d) * (1 / dir_d);
float  t_f   = (b - orig) * (1 / dir);

float X = -orig * (1/dir);
float Y = (1/dir);
float t_fma  = FMA(b, y, X); // b*Y + X;
printf("double(reference)=%.15g, float=%.15g, FMA=%.9g\n",
       t_ref, t_f, t_fma);
// outputs:  double(reference)=0.193715096009103,
//           float            =0.19371509552002, FMA=0.18670845
}
```

Listing 2. Proof by example that the FMA version is susceptible to large amounts of error.

```
template <typename IN_T, typename OUT_T>
inline OUT_T reinterpret_type(const IN_T in) {
   // Good compiler should optimize memcpy away.
   OUT_T out;
   memcpy(&out, &in, sizeof(out));
   return out;
 }

inline float add_ulp_magnitude(float f, int ulps) {
  if (!std::isfinite(f)) return f;
  const unsigned bits = reinterpret_type<float, unsigned>(f);
  return reinterpret_type<unsigned, float>(bits + ulps);
}
```

Listing 3. Increasing magnitude of a float by a certain number of ulps. A double precision version would only require changing the `float`s to `double`s and `bits` to be `unsigned long long`.

each time a ray-box test occurs. While this works, it is somewhat inefficient as it must occur for each ray-bounding box test, and on certain hardware architectures, there can be even more overhead, as adding ulps requires moving the float from a floating-point register into an integer register and back. Note that the code in Listing 3 will not work for floats that are less than k ulps from `FLT_MAX` when adding k ulps to the float, since then we could obtain NaN or some incorrect number. If this is an issue, this can be easily avoided by clamping to inf at the expense of some additional overhead.

4.1. Inverse ulps traversal

Let us define $ray_{\text{inv\_dir}} = \frac{1}{ray_{\text{dir}}}$ and reorder the sequence of operations in Equation (5) so that we get the equivalent:

$$\tilde{t} = \left(b - ray_{\text{origin}}\right)\left(ray_{\text{inv\_dir}}\left(1+\delta_1+\delta_2+\delta_3\right)\right) \tag{26}$$

The error is maximized when

$$\tilde{t} = \left(b - ray_{\text{origin}}\right)\left(ray_{\text{inv\_dir}}\left(1\pm 3\varepsilon\right)\right) \tag{27}$$

A nice property, easily seen in Equation (27), is that the $\frac{3}{2}$ ulps can be applied to the inverse ray direction instead of to the final t. So we can also arrive at Equation (25), not by adding two ulps to t, but by adding two ulps to $ray_{\text{inv\_dir}}$. This allows us to precompute a maximum error $\widetilde{ray}_{\text{inv\_dir}}$ once and then reuse that for all bounding-box tests with that ray when computing $\tilde{t}_{max}$. The final modified algorithm based on Williams et al. and Berger-Perrin is presented in Listing 4. In this paper, we will call this algorithm the InvUlps traversal. Its benefits are that it adds only the minimum amount of padding, so introduces as few false-hits as possible, is fast to precompute, and no additional computation is introduced during the traversal loop. However, extra work might occur during the traversal loop in the form of additional loads and register pressure, which could make it slightly slower on some architectures.

The implementation from Listing 3 works correctly here for all floats f, since $1/f$ is either inf or at least 7 ulps below `FLT_MAX`, so no overflow to NaN can occur when adding two ulps.

4.2. Max multiplication traversal

If we allow adding slightly more padding than required, a very simple solution is to approximate Equation (25) with a conservative multiplication that adds at least 2 ulps.

$$\tilde{t}_{max\_padded} = \tilde{t}_{max} + 2ulp(\tilde{t}_{max}) \tag{28}$$
$$\tilde{t}_{max\_padded} = \tilde{t}_{max} + 2(2\varepsilon\tilde{t}_{max}) \tag{29}$$
$$\tilde{t}_{max\_padded} = \tilde{t}_{max}(1+4\varepsilon) \tag{30}$$
$$\tilde{t}_{max\_padded} = \tilde{t}_{max}1.00000023841857910156 25 \tag{31}$$
$$\tilde{t}_{max\_padded} = \tilde{t}_{max}\texttt{1.00000024f}, \tag{32}$$

where we used the ε for 32 bit floats. Note that `1.00000023841857910156 25` and `1.00000024f` are exactly equal since $1+4\varepsilon$ happens to be a number that can be perfectly represented in floating-point and `1.00000024f` rounds to this value. Since Equation (1) gives a lower bound, it is possible that more than 2 ulps will be added. In fact, we confirmed by exhaustively testing all normalized floats that it will add between 2 and 4 actual ulps. Note that this is valid for all normalized floats ($\tilde{t}_{max} >= 2^{-126}$)

```
struct Ray {
  Ray(Vector3 &o, Vector3 &d) : origin(o), dir(d) {
    inv_dir = Vector3(1/d.x, 1/d.y, 1/d.z);
    inv_dir_pad.x = add_ulp_magnitude(inv_dir.x, 2);
    inv_dir_pad.y = add_ulp_magnitude(inv_dir.y, 2);
    inv_dir_pad.z = add_ulp_magnitude(inv_dir.z, 2);
    sign[0] = (inv_dir.x < 0);
    sign[1] = (inv_dir.y < 0);
    sign[2] = (inv_dir.z < 0);
  }
  Vector3 origin;
  Vector3 dir;
  Vector3 inv_dir;
  Vector3 inv_dir_pad;
  int sign[3];
};

bool Box::intersect_InvUlps(const Ray &r, float tmin, float tmax) {
  float txmin, txmax, tymin, tymax, tzmin, tzmax;
  txmin = (bounds[  r.sign[0]].x-r.origin.x) * r.inv_dir.x;
  txmax = (bounds[1-r.sign[0]].x-r.origin.x) * r.inv_dir_pad.x;
  tymin = (bounds[  r.sign[1]].y-r.origin.y) * r.inv_dir.y;
  tymax = (bounds[1-r.sign[1]].y-r.origin.y) * r.inv_dir_pad.y;
  tzmin = (bounds[  r.sign[2]].z-r.origin.z) * r.inv_dir.z;
  tzmax = (bounds[1-r.sign[2]].z-r.origin.z) * r.inv_dir_pad.z;
  tmin = max(tzmin, max(tymin, max(txmin, tmin)));
  tmax = min(tzmax, min(tymax, min(txmax, tmax)));
  return tmin <= tmax;
}
```

Listing 4. InvUlps robust BVH traversal. Changes from the default algorithm are highlighted.

as well as for NaN and infs. Not being robust for denormals is an issue when the ray exits the bounds with $t < 2^{-126}$; however, most ray tracers do not allow intersections within some range of the ray origin and that range is extremely likely to be greater than this (it is usually larger than ε), so missing this bounding box would be fine since nothing could have been intersected in it anyways. Even if no range is used, it is extremely unlikely that the ray would hit something valid so close from its origin. If this must be allowed, then the following would handle all floating-point values, $\tilde{t}_{max}$`1.00000024f` + `2.80259693e-45f`, but would be slightly slower due to the extra addition and possibly much slower on architectures where denormals are expensive.

Unlike adding an ulp at each traversal step, which can be slow on certain hardware architectures, multiplying will always be fast and is an extremely simple modification to existing BVH traversal code, as seen in Listing 5. We call this the MaxMult traversal. Furthermore, we do not need to compute the padded inverse ray direction

```
struct Ray {
  Ray(Vector3 &o, Vector3 &d) : origin(o), dir(d) {
    inv_dir = Vector3(1/d.x, 1/d.y, 1/d.z);
    sign[0] = (inv_dir.x < 0);
    sign[1] = (inv_dir.y < 0);
    sign[2] = (inv_dir.z < 0);
  }
  Vector3 origin;
  Vector3 dir;
  Vector3 inv_dir;
  int sign[3];
};

bool Box::intersect_MaxMult(const Ray &r, float tmin, float tmax) {
  float txmin, txmax, tymin, tymax, tzmin, tzmax;
  txmin = (bounds[  r.sign[0]].x-r.origin.x) * r.inv_dir.x;
  txmax = (bounds[1-r.sign[0]].x-r.origin.x) * r.inv_dir.x;
  tymin = (bounds[  r.sign[1]].y-r.origin.y) * r.inv_dir.y;
  tymax = (bounds[1-r.sign[1]].y-r.origin.y) * r.inv_dir.y;
  tzmin = (bounds[  r.sign[2]].z-r.origin.z) * r.inv_dir.z;
  tzmax = (bounds[1-r.sign[2]].z-r.origin.z) * r.inv_dir.z;
  tmin = max(tzmin, max(tymin, max(txmin, tmin)));
  tmax = min(tzmax, min(tymax, min(txmax, tmax)));
  tmax *= 1.00000024f;
  return tmin <= tmax;
}
```

Listing 5. MaxMult robust BVH traversal. By allowing for up to 4 ulps instead of 2 ulps of padding, the following code is robust, efficient, and extremely simple to implement. Changes from the original algorithm are highlighted. For double precision, the multiplication factor is `1.0000000000000004` instead.

like in InvUlps, so if few traversals occur, this might even be faster. The only significant downsides are that it is often 2 ulps too large, which is usually a minor issue, and it does add a small amount of extra work during each traversal step.

5. Results

All our tests were made with Solid Angle's Arnold renderer on 8 cores of an Intel Xeon E5-4650 running at 2.7GHz. We used a high quality double precision triangle-intersection algorithm so that false-misses caused by the triangle intersection test would be kept at a minimum. Two scenes are from test cases reported to us by our users. One is a poppy flower from the film "Oz: the Great and Powerful" (Figure 2), and the other is the staircase used in the Dragon Age 2 cinematic trailer (Figure 4). False misses with these models were causing problems in the final rendered images, showing

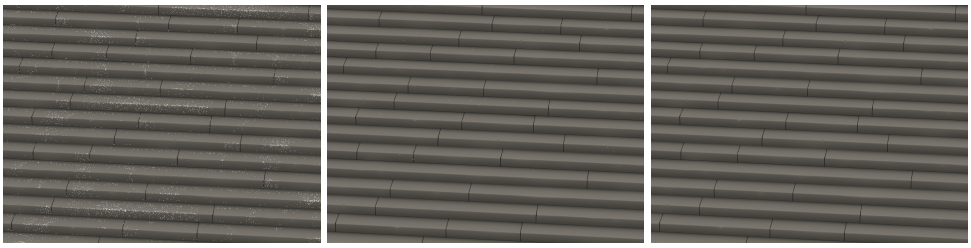

Figure 4. We rendered the 90.1M triangle stairs at a 960×720 resolution with 256 samples per pixel using an ambient occlusion shader. From left to right, we used no padding of any kind, one, and then two ulps in our robust algorithm. Note that using one ulp still results in false-misses, while two ulps renders correctly without holes.

that false-misses are a legitimate issue and not just one of academic interest. We have modified these test cases by further subdividing the meshes and casting more rays per pixel in an attempt to cause many more false-misses to occur, which in turn gives us a better test as to whether we are in fact preventing all false-misses. Furthermore, to aid with visualizing these misses, we have simplified the shading and placed extremely bright backgrounds behind the model so that even a single false-miss will clearly turn up in the final image. We also compare against the publicly available Natural History Museum scene (Figure 5), which does not exhibit any perceptible artifacts in our renderings, although an image comparison tool does single out a few pixels as having a very slight difference. Despite not easily showing artifacts, this scene helps to compare the performance of our algorithms on more realistic looking scenes with greater geometric complexity.

In the flower scene, the camera is 66400 units away from the flower which is located roughly at the origin and is 29 units tall. Extreme distances, such as this one, are especially challenging, since it amplifies numerical precision issues. Of course, in this particular test case, the camera could have been moved much closer and fewer holes would have shown up. However, in the film, this flower might be part of a vast field, and so, a large camera distance is required. The stairs, by comparison, is at a more regular distance of between 408 and 454 units from the camera, and the museum has the camera inside it with the longest dimension of the scene being about 140 units.

Since we need to expand the entry and exit planes by two ulps, an alternative (third) solution to prevent false-misses is to add the padding not to the inverse ray direction but instead to the leaf bounds by adding one $ulp(b - ray_{\text{origin}}))$ in each positive and negative direction during BVH construction. Notice that this padding depends on the variable ray origin, so we need to select the furthest possible ray origin from the bounding box in order to guarantee there will never be a false-miss. Since the stairs are at most 454 units away from the camera, we pad the bounding boxes by $ulp(454) = 2\varepsilon 454 = 2^{-23} * 454 = 5.412 \times 10^{-5}$ so that ray origins can exist within a

Figure 5. We rendered the 1.4M triangle Natural History Museum at a 960×720 resolution with 256 samples per pixel using an ambient occlusion shader to demonstrate that no perceptible errors arising from precision limitations are present.

range of 454 units from the object. Table 1 shows that, while this is competitive, it is the slowest option and has the most traversal overhead. Worse still, over-padding by 1×10^{-3} makes it a further couple percent slower at 72.9s, and since this padding must be set during BVH construction, this means that there is a trade-off between performance and how far away the camera and other objects and lights are allowed to be.

The Museum scene, with longest dimension being 140 units, requires padding to about 1.67×10^{-6}. Despite having a small amount of padding, the padding causes a 3% slowdown and a significant number of extra traversals and intersections.

Similarly, for the flower, we must pad by $ulp(66400) = 0.00792$ and find that padding the bounds is much slower than the alternative methods, as seen in Table 1. In this test, performance and visual correctness are quite sensitive to the amount of padding. With a padding of 0.01, which is just $1.3\times$ larger, the render time goes up by an additional $1.2\times$, while a smaller padding of 0.0031, which renders at 106.6s, already exhibits holes. The slow performance occurs because we are forcing all rays, even rays that are near the object and could get by with much smaller bounding box padding (such as the ambient occlusion rays), to use the same overly large padding.

	no padding time trav/int	bounds padding time trav/int	InvUlp time trav/int	MaxMult time trav/int
Flower	78.0s 1×/1×	174.4s 1.323×/6.600×	79.6s 1.013×/1.077×	81.2s 1.018×/1.108×
Stairs	70.8s 1×/1×	71.3 1.004×/1.035×	71.2s 1.000×/1.002×	71.0s 1.000×/1.002×
Museum	114.1s 1×/1×	117.3s 1.022×/1.119×	114.1s 1.000×/1.002×	113.4s 1.000×/1.002×

Table 1. Table shows rendering time and relative number of BVH traversals (*trav*) and triangle intersection tests (*int*) compared to the default non-robust traversal where no padding is used. From left to right, the columns correspond to: using no padding so that false-misses occur, padding the bounds of every BVH node by the minimum fixed epsilon specified during BVH construction, adding exactly 2 ulps to the padded inverse direction when the ray is created, and finally, multiplying $\tilde{t}_{max}$ by `1.00000024f` so that at least 2 ulps of padding are added to $\tilde{t}_{max}$ at each traversal step.

This demonstrates how important it is to accurately compute the padding and how there can be an arbitrarily large range in what the required padding must be. Even using the minimal padding for a scene can still be slow. We are not aware of any published work that gives an actual formula for the minimum padding to use, and most likely, ray tracers that pad the BVH bounds have a hard coded padding or some other heuristic that tends to over-pad and/or under-pad, so the above formula is already a substantial improvement over the current state-of-the-art. Aside from the potential render-time overhead, a major limitation of this approach is that it requires knowing the maximum distance a ray will travel. Computing this maximum distance can be complicated, especially in the case of object instancing, since then the maximum distance for the furthest instance must be used over all instances (which is a further unneeded performance hit for near instances), and with lazy loading, the position of all instances might not even be known yet. This is also constraining for interactive renderers, which would have to guess at a maximum distance and then constrain the user from not exceeding it.

The two solutions we recommend are InvUlps, which pads the inverse ray direction by 2 ulps, and MaxMult, which multiplies $\tilde{t}_{max}$ by `1.00000024f` so that at least 2 ulps of padding are added. Table 1 shows that both methods essentially introduced no overhead in the stairs and museum scene and only a slight amount of overhead in the Flower. Most of this overhead is due to the flower being very far away, which makes the distance an ulp covers much larger, and so, more nodes end up being traversed and more triangles tested against. In this case, the up to two extra ulps of padding that MaxMult must do compared to InvUlps makes it traverse more nodes and consequently

become slower.

6. Discussion

This paper shows only how to make the BVH traversal robust, but there can still be sources of error in other parts of the ray tracer that can lead to holes or other artifacts. A common case is that holes and self shadowing can appear along the shared edge of two triangles if the intersection algorithm is not robust enough. Another issue is that if the minimum and maximum t values passed to the bounding box intersection function are incorrect, then the wrong primitive could be intersected. One way this can occur is if two axis-aligned triangles, A and B, are nearly coplanar, with A being k ulps in front of B, and each is in a separate flat bounding box. During BVH traversal, assume that B's bounding box is tested first and B is then hit and due to numerical precision issues in the ray-triangle intersection, its ray-primitive intersection is computed as $\tilde{t} = t - (k+1)ulp(t)$. This $k+1$ ulps of error ends up placing B erroneously in front of A. Early ray termination using that intersection distance $\tilde{t}$ would then cause the bounding box of A to be skipped, since A is behind the computed hit point. We do not count this as an error on the part of the BVH traversal, since it was given the wrong t-bounds to traverse. Having said that, this could be fixed by performing early-ray termination not with $\tilde{t}$ but with $\tilde{t} + \alpha$, where α is the maximum error that could occur from that primitive-ray intersection. Computing this α for a particular ray-primitive algorithm would be the domain of a future paper.

7. Conclusion

We have derived how numerical precision issues can occur during BVH traversal, and from this derivation, we were able to come up with two elegant solutions—InvUlps and MaxMult—towards making BVH traversal fully robust while introducing essentially negligible overhead in both code complexity and performance. Which of these two methods is fastest will likely be scene and hardware specific. For instance, InvUlps could be slow if adding ulps is expensive on that hardware, and the three extra padded inverse ray direction components cause register spilling to occur. Since MaxMult pads by up to 2 more ulps than necessary, this can make it slower than InvUlps if the parametric ray distance is extremely large, such as with the flower. However, in most scenes, this is not the case, and so, is a non-issue.

We tested this for single precision floats, but this can be trivially extended to lower or higher precision floats as well, for instance, for double precision we would multiply by `1.0000000000000004` and for 16-bit floats we would use `1.001953125`. We also gave examples of how commonly used solutions towards handling false-misses, such as padding bounding boxes or casting more samples, will not always work, and even if it does work, it can result in a significant performance hit.

Acknowledgements

We would like to thank Digic Pictures for the staircase model and Sony Pictures Imageworks for the flower model. The Natural History Museum scene was modeled by Alvaro Luna Bautista and comes from the 3dRender.com Lighting Challenges. Chris Kulla, Marcos Fajardo, and Alan King provided numerous helpful suggestions and comments for which we are thankful. Sean Barrett pointed out that the original FMA error analysis was incorrect.

References

BERGER-PERRIN, T., 2004. SSE ray/box intersection test. http://www.flipcode.com/archives/SSE_RayBox_Intersection_Test.shtml. 3

KAY, T., AND KAJIYA, J. 1986. Ray tracing complex scenes. In *Proceedings of SIGGRAPH*, 269–278. 3

MAHOVSKY, J. 2005. *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, University of Calgary. 1

WHITEHEAD, N., AND FIT-FLOREA, A., 2011. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA white paper. 4, 6

WILLIAMS, A., BARRUS, S., MORLEY, R. K., AND SHIRLEY, P. 2005. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools 10*, 1, 49–54. 3

Thiago Ize, Robust BVH Ray Traversal, *Journal of Computer Graphics Techniques (JCGT)*, vol. 0, no. 0, 0–0, 2015
http://jcgt.org/published/0002/02/02/

Received:	2014-10-24		
Recommended:	pending review	Corresponding Editor:	
Published:	pending review	Editor-in-Chief:	Morgan McGuire