tavianator.com

PROGRAMMING, RAY TRACING

# FAST, BRANCHLESS RAY/BOUNDING BOX INTERSECTIONS, PART 2: NANS

MARCH 23, 2015 | TAVIAN BARNES | 7 COMMENTS

In part 1, I outlined an algorithm for computing intersections between rays and axis-aligned bounding boxes. The idea to eliminate branches by relying on IEEE 754 floating point properties goes back to Brian Smits in [1], and the implementation was fleshed out by Amy Williams. et al. in [2].

To quickly recap, the idea is to replace the naïve slab method:

```
bool intersection(box b, ray r) {
    double tmin = -INFINITY, tmax = INFINITY;

    for (int i = 0; i < 3; ++i) {
        if (ray.dir[i] != 0.0) {
            double t1 = (b.min[i] - r.origin[i])/r.dir[i];
            double t2 = (b.max[i] - r.origin[i])/r.dir[i];

            tmin = max(tmin, min(t1, t2));
            tmax = min(tmax, max(t1, t2));
        } else if (ray.origin[i] <= b.min[i] || ray.origin[i] >= b.max[i]) {
            return false;
        }
    }

    return tmax > tmin && tmax > 0.0;
}
```

with the equivalent but faster:

```
bool intersection(box b, ray r) {
    double tmin = -INFINITY, tmax = INFINITY;

    for (int i = 0; i < 3; ++i) {
        double t1 = (b.min[i] - r.origin[i])*r.dir_inv[i];
        double t2 = (b.max[i] - r.origin[i])*r.dir_inv[i];

        tmin = max(tmin, min(t1, t2));
        tmax = min(tmax, max(t1, t2));
    }
```

```
    return tmax > max(tmin, 0.0);
}
```

Are the two algorithms really equivalent? We've eliminated the $\texttt{ray.dir}_i \neq 0$ checks by relying on IEEE 754 floating point behaviour. When $\texttt{ray.dir}_i = \pm 0$, $\texttt{ray.dir\_inv}_i = \pm\infty$. If the ray origin's $i$ coordinate is inside the box, meaning $\texttt{b.min}_i < \texttt{r.origin}_i < \texttt{b.max}_i$, we'll have $t_1 = -t_2 = \pm\infty$. Since $\max(n, -\infty) = \min(n, +\infty) = n$ for all $n$, $t_{\min}$ and $t_{\max}$ will remain unchanged.

On the other hand, if the $i$ coordinate is outside the box ($\texttt{r.origin}_i < \texttt{b.min}_i$ or $\texttt{r.origin}_i > \texttt{b.max}_i$), we'll have $t_1 = t_2 = \pm\infty$, and therefore either $t_{\min} = +\infty$ or $t_{\max} = -\infty$. One of those values will stick around through the rest of the algorithm, causing us to return `false`.

Unfortunately the above analysis has a caveat: if the ray lies exactly on a slab ($\texttt{r.origin}_i = \texttt{b.min}_i$ or $\texttt{r.origin}_i = \texttt{b.max}_i$), we'll have (say)

$$\begin{aligned} t_1 &= (\texttt{b.min}_i - \texttt{r.origin}_i) \cdot \texttt{r.dir\_inv}_i \\ &= 0 \cdot \infty \\ &= \text{NaN} \end{aligned}$$

which behaves a lot less nicely than infinity. Correctly handling this edge (literally!) case depends on the exact behaviour of `min()` and `max()`.

## On min and max

The most common implementation of `min()` and `max()` is probably

```
#define min(x, y) ((x) < (y) ? (x) : (y))
#define max(x, y) ((x) > (y) ? (x) : (y))
```

This form is so pervasive it was canonised as the behaviour of the min/max instructions in the SSE/SSE2 instruction sets. Using these instructions is key to getting good performance out of the algorithm. That being said, this form has some odd behaviour involving NaN. Since all comparisons with NaN are false,

$$\begin{aligned} \min(x, \text{NaN}) &= \max(x, \text{NaN}) &= \text{NaN} \\ \min(\text{NaN}, x) &= \max(\text{NaN}, x) &= x. \end{aligned}$$

The operations neither propagate nor suppress NaNs; instead, when either argument is NaN, the second argument is always returned. (There is also similar odd behaviour concerning signed zero, but it doesn't affect this algorithm.)

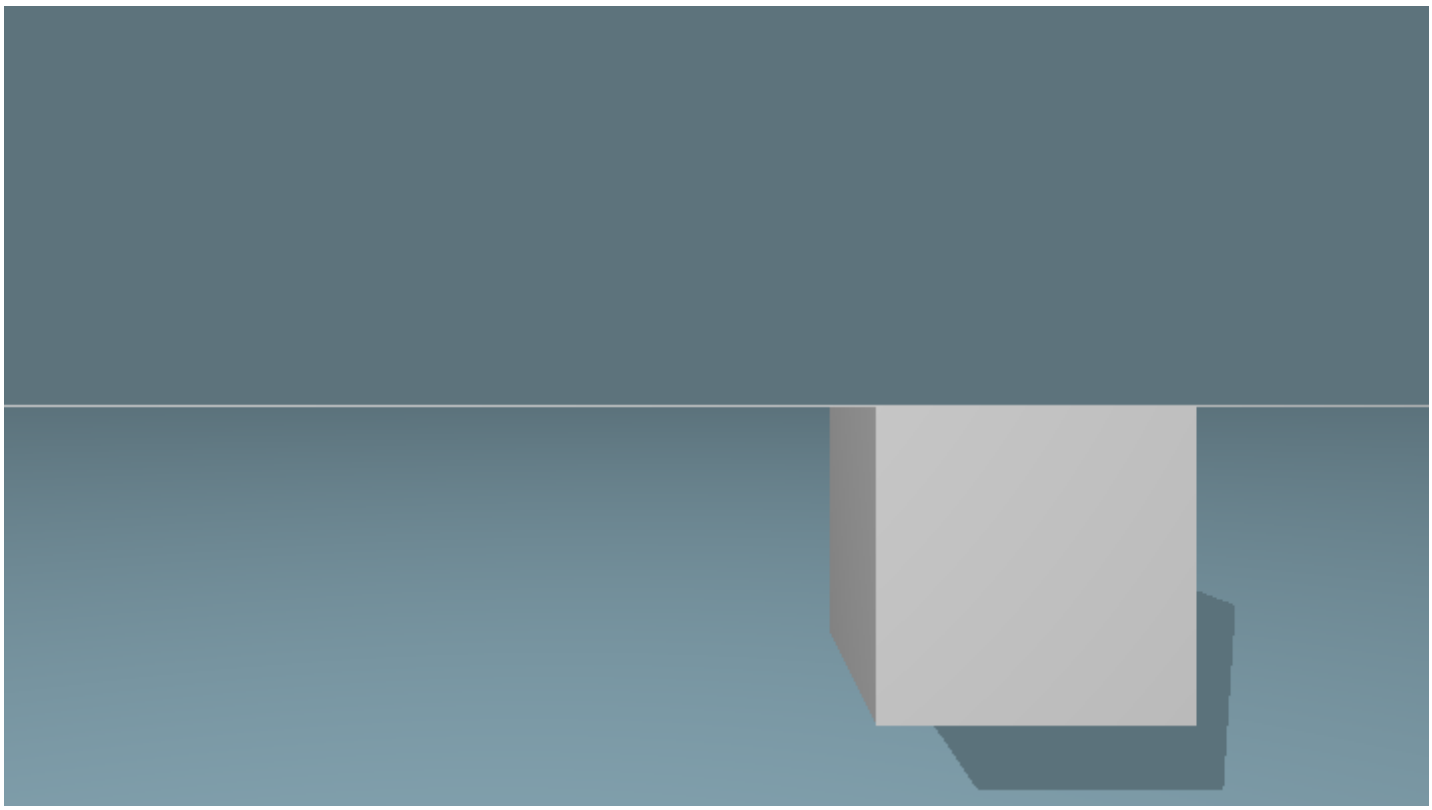
In contrast, the IEEE 754-specified min/max operations (called "minNum" and "maxNum") suppress NaNs, always returning a number if possible. This is also the behaviour of C99's `fmin()` and `fmax()` functions. On the other hand, Java's `Math.min()` and `Math.max()` functions propagate NaNs, staying consistent with most other binary operations on floating point values. [3] and [4] have some more discussion about the various min/max implementations in the wild.

## The problem

The IEEE and Java versions of `min()` and `max()` provide consistent behaviour: all rays that lie exactly on a slab are considered not to intersect the box. It's easy to see why for the Java version, as the NaNs eventually pollute all the computations and make us return `false`. For the IEEE version, $\min(t_1, t_2) = \max(t_1, t_2) = \pm\infty$, which is the same as when the ray is entirely outside the box.

(Since this is an edge case, you might wonder why we don't choose to return `true` instead of `false` for rays on the boundary. It turns out to be much harder to get this behaviour with efficient code.)

With the SSE-friendly min/max implementations, the behaviour is inconsistent. Some rays that lie on a slab will intersect, even if they are completely outside the box in another dimension:



In the above image, the camera lies in the plane of the top face of the cube, and intersections are computed with the above algorithm. The top face is seen to extend out past the sides, due to improper NaN handling.

## The workaround

When at most one argument is NaN, we can simulate the IEEE behaviour with

$$\begin{aligned}\mathrm{minNum}(x, y) &= \min(x, \min(y, \infty)) \\ \mathrm{maxNum}(x, y) &= \max(x, \max(y, -\infty))\end{aligned}$$

Thierry Berger-Perrin applies a similar strategy in [5], effectively computing

```
tmin = max(tmin, min(min(t1, t2), INFINITY));
tmax = min(tmax, max(max(t1, t2), -INFINITY));
```

in the loop instead. It is also fine to do

```
tmin = max(tmin, min(min(t1, t2), tmax));
tmax = min(tmax, max(max(t1, t2), tmin));
```

which is around 30% faster because CPUs are slower at handling floating point special cases (infinities, NaNs, subnormals). For the same reason, it's better to unroll the loop like this, to avoid dealing with any more infinities than necessary:

```
bool intersection(box b, ray r) {
    double t1 = (b.min[0] - r.origin[0])*r.dir_inv[0];
    double t2 = (b.max[0] - r.origin[0])*r.dir_inv[0];

    double tmin = min(t1, t2);
    double tmax = max(t1, t2);

    for (int i = 1; i < 3; ++i) {
        t1 = (b.min[i] - r.origin[i])*r.dir_inv[i];
        t2 = (b.max[i] - r.origin[i])*r.dir_inv[i];

        tmin = max(tmin, min(min(t1, t2), tmax));
        tmax = min(tmax, max(max(t1, t2), tmin));
    }

    return tmax > max(tmin, 0.0);
}
```

It's a little harder to see why this version is correct: any NaNs from the $x$ coordinate will propagate through to the end, while NaNs from other coordinates will result in $t_{\min} \geq t_{\max}$; in both cases, `false` is returned.

With GCC 4.9.2 at `-O3` this implementation handles just over 93 million rays per second, meaning the runtime is around 30 clock cycles, even without vectorization!

## The other workaround

Sadly, that's still about 15% slower than the version with no explicit NaN handling. And since this algorithm is generally used when traversing a bounding volume hierarchy, the worst thing that can happen is you traverse more nodes than necessary in degenerate cases. For many applications, this is well worth it, and it should never result in any visual artifacts in practice if the ray/object intersection functions are implemented correctly. For completeness, here's a fast implementation (108 million rays/second) that doesn't attempt to handle NaNs consistently:

```
bool intersection(box b, ray r) {
    double t1 = (b.min[0] - r.origin[0])*r.dir_inv[0];
    double t2 = (b.max[0] - r.origin[0])*r.dir_inv[0];

    double tmin = min(t1, t2);
    double tmax = max(t1, t2);

    for (int i = 1; i < 3; ++i) {
        t1 = (b.min[i] - r.origin[i])*r.dir_inv[i];
        t2 = (b.max[i] - r.origin[i])*r.dir_inv[i];

        tmin = max(tmin, min(t1, t2));
        tmax = min(tmax, max(t1, t2));
    }

    return tmax > max(tmin, 0.0);
}
```

The program I used to test various `intersection()` implementations is given in [6]. In my next post on this topic, I'll talk about low-level implementation details, including vectorization, to get the most performance possible out of this algorithm.

[1]: Brian Smits: Efficiency Issues for Ray Tracing. *Journal of Graphics Tools* (1998).
[2]: Amy Williams. et al.: An Efficient and Robust Ray-Box Intersection Algorithm. *Journal of Graphics Tools* (2005).
[3]: https://groups.google.com/forum/#!topic/llvm-dev/-SKl0nOJW_w
[4]: https://ghc.haskell.org/trac/ghc/ticket/9251
[5]: http://www.flipcode.com/archives/SSE_RayBox_Intersection_Test.shtml
[6]: https://gist.github.com/tavianator/132d081ed4d410c755fd

## 7 THOUGHTS ON "FAST, BRANCHLESS RAY/BOUNDING BOX INTERSECTIONS, PART 2: NANS"

**Sven-Hendrik Haase**

NOVEMBER 20, 2015 AT 12:36 AM

For some reason, your code doesn't yield any results in my program. I'm using glm and wrote it like this: https://gist.github.com/a64045811d5dcf378b6a

It just never collides with anything. However, this: https://gist.github.com/a64045811d5dcf378b6a from scratchapixel.com works just fine.

I would like to use your algorithm, though, since it's likely faster. Can you help me figure out where I went wrong?

**★ Tavian Barnes**

DECEMBER 1, 2015 AT 8:32 PM

Well, here, you are multiplying by `dir.x` instead of `dir_inv.x`.

**Sven-Hendrik Haase**

DECEMBER 3, 2015 AT 3:06 AM

Oh gee, thanks man! Awesome work.

**Anna**

FEBRUARY 8, 2017 AT 2:15 PM

Hi! Why is this article using "return tmax > tmin" rather then "return tmax >= tmin" from the previous article?

**★ Tavian Barnes**

FEBRUARY 18, 2017 AT 3:36 PM

You will have tmax == tmin whenever the ray exactly intersects and edge or corner of the bounding box. Since we already decided that a ray that lies exactly in the plane of a face isn't an intersection, I thought it made sense for the edges and corners to not count either.

**Matas Peciukonis**

MARCH 26, 2017 AT 4:52 PM

How do you get the normals of the box, I don't understand, without them , how do you shade anything?

**★ Tavian Barnes**

MARCH 29, 2017 AT 2:08 PM

This is for bounding boxes, not box objects in a scene. But if you want to use this approach for boxes in your scene, you can compute the intersection point in object space (assuming your box is

from (-1, -1, -1) to (1, 1, 1)) and then clip all the values that aren't very close to -1 or 1 to zero. For example, if the intersection point is (-1, 0.1, 0.9), the normal is (-1, 0, 0).