tavianator.com

PROGRAMMING, RAY TRACING

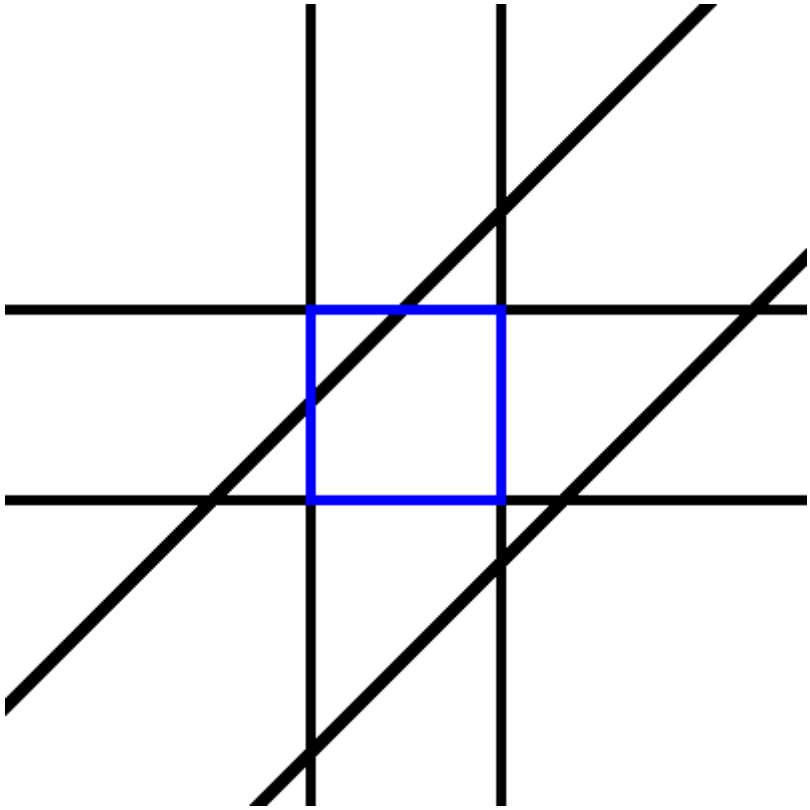
FAST, BRANCHLESS RAY/BOUNDING BOX INTERSECTIONS

MAY 2, 2011 | TAVIAN BARNES | 29 COMMENTS

(Update: part 2)

Axis-aligned bounding boxes (AABBs) are universally used to bound finite objects in ray-tracing. Ray/AABB intersections are usually faster to calculate than exact ray/object intersections, and allow the construction of bounding volume hierarchies (BVHs) which reduce the number of objects that need to be considered for each ray. (More on BVHs in a later post.) This means that a ray-tracer spends a lot of its time calculating ray/AABB intersections, and therefore this code ought to be highly optimised.

The fastest method for performing ray/AABB intersections is the slab method. The idea is to treat the box as the space inside of three pairs of parallel planes. The ray is clipped by each pair of parallel planes, and if any portion of the ray remains, it intersected the box.



A simple implementation of this algorithm might look like this (in two dimensions for brevity):

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

    if (ray.n.x != 0.0) {
        double tx1 = (b.min.x - r.x0.x)/r.n.x;
        double tx2 = (b.max.x - r.x0.x)/r.n.x;

        tmin = max(tmin, min(tx1, tx2));
        tmax = min(tmax, max(tx1, tx2));
    }

    if (ray.n.y != 0.0) {
        double ty1 = (b.min.y - r.x0.y)/r.n.y;
        double ty2 = (b.max.y - r.x0.y)/r.n.y;

        tmin = max(tmin, min(ty1, ty2));
        tmax = min(tmax, max(ty1, ty2));
    }

    return tmax >= tmin;
}
```

However, those divisions take quite a bit of time. Since when ray-tracing, the same ray is tested against many AABBs, it makes sense to pre-calculate the inverses of the direction components of the ray. If we can rely on the IEEE 754 floating-point properties, this also implicitly handles the edge case where a component of the direction is zero - the `tx1` and `tx2` values (for example) will be infinities of opposite sign if the ray is within the slabs, thus leaving `tmin` and `tmax` unchanged. If the ray is outside the slabs,

`tx1` and `tx2` will be infinities with the same sign, thus making `tmin == +inf` or `tmax == -inf`, and causing the test to fail.

The final implementation would look like this:

```
bool intersection(box b, ray r) {
    double tx1 = (b.min.x - r.x0.x)*r.n_inv.x;
    double tx2 = (b.max.x - r.x0.x)*r.n_inv.x;

    double tmin = min(tx1, tx2);
    double tmax = max(tx1, tx2);

    double ty1 = (b.min.y - r.x0.y)*r.n_inv.y;
    double ty2 = (b.max.y - r.x0.y)*r.n_inv.y;

    tmin = max(tmin, min(ty1, ty2));
    tmax = min(tmax, max(ty1, ty2));

    return tmax >= tmin;
}
```

Since modern floating-point instruction sets can compute min and max without branches, this gives a ray/AABB intersection test with no branches or divisions.

My implementation of this in my ray-tracer Dimension can be seen here.

29 THOUGHTS ON “FAST, BRANCHLESS RAY/BOUNDING BOX INTERSECTIONS”

Phil

APRIL 1, 2012 AT 5:24 AM

This doesn't seem to work out for me for negative direction vectors. Looking forward, I can see a box that's there, but looking backwards, I again see the same box mirrored, even though in this direction it's not "there". Following the algorithm step by step manually for both a positive and a negative z-dir (with x-dir and y-dir set to 0) gives the same near and far planes in both directions:

```
/*
box = MIN 0,0,0  MAX 256,256,256
ray POS 128,128,-512
case 1: ray DIR 0,0,0.9 -- inverse: (inf,inf,1.1111)
case 2: ray DIR 0,0,-0.9 -- inverse: (inf,inf,-1.1111)
*/
picker.tx1, picker.tx2 = (me.Min.X - ray.Pos.X) * ray.invDir.X, (me.Max.X - ray.Pos.
picker.ty1, picker.ty2 = (me.Min.Y - ray.Pos.Y) * ray.invDir.Y, (me.Max.Y - ray.Pos.
picker.tz1, picker.tz2 = (me.Min.Z - ray.Pos.Z) * ray.invDir.Z, (me.Max.Z - ray.Pos.
picker.txn, picker.txf = math.Min(picker.tx1, picker.tx2), math.Max(picker.tx1, pick
picker.tyn, picker.tyf = math.Min(picker.ty1, picker.ty2), math.Max(picker.ty1, pick
picker.tzn, picker.tzf = math.Min(picker.tz1, picker.tz2), math.Max(picker.tz1, pick
```

```
picker.tnear = math.Max(picker.txn, math.Max(picker.tyn, picker.tzn))
picker.tfar = math.Min(picker.txf, math.Min(picker.tyf, picker.tzf))
if picker.tfar < picker.tnear {
        return true
}
```

★ tavianator

APRIL 3, 2012 AT 2:58 PM

Right, because the test is only for whether the line intersects the box at all. The line extends both forwards and backwards. Just add a tmax >= 0 check. It's tmax, not tmin, since tmin will be < 0 if the ray originates inside the box.

Sergiy

DECEMBER 30, 2012 AT 7:42 AM

Thanks. Works nicely and fast. I updated it a bit, to use SSE (though Vectormath), floats only.

https://gist.github.com/4412640#file-bbox-cpp-L14

★ tavianator

JANUARY 14, 2013 AT 9:34 AM

You're welcome! I can't see that gist though (says "OAuth failure"). What kind of performance did the vectorisation give you?

Bram Stolk

DECEMBER 29, 2014 AT 5:30 PM

To get the actual intersection, would I just use tmin, and multiply it with ray dir, adding ray origin? And what if I'm just interested in which face was intersected? x+,x-,y+,y-,z+ or z-?

★ tavianator

JANUARY 11, 2015 AT 4:22 PM

Yes, that's what I'd do. Except if the ray origin is inside the box (`tmin < 0`), you need to use `tmax` instead.

To see what face was intersected, there's a few different ways. You can keep track of which slab is intersecting in the above algorithm at all times, but that slows it down.

For a cube centered at the origin, a neat trick is to take the component of the intersection point with the largest absolute value.

Francisco

MARCH 12, 2015 AT 12:30 PM

So if I return a vec3 with tmin, tmax and a float hit = step(tmin,tmax)*step(0,tmax), I basically know that if hit > 0 then ro+rd*tmin is my lower bound intersection point (entry point) and ro+rd*tmax is my higher bound intersection point (exit point), right? However, if tmin < 0, I'm inside the bounding box, which means I don't need the entry point and I can just use the ray origin as my starting point.

★ Tavian Barnes

MARCH 12, 2015 AT 2:11 PM

Sorry, not sure what you mean by `step()`. You can get the starting point as `ro + td*max(tmin, 0.0)`.

Francisco

APRIL 1, 2015 AT 12:20 PM

step(edge,A) returns 1 if A >= edge, 0 elsewise. So I know if it was a hit if tmax >= tmin AND tmax >= 0. Then we also know tmin is the closest intersection and tmax is the furthest.

I implemented this in C++ and it works perfectly for any ray direction. However, in GLSL it seems to have problems with negative directions.

Bram Stolk

JANUARY 11, 2015 AT 5:18 PM

Thanks Tavianator,

Yeah, the cube-trick requires sorting.
So for now I test proximity to face within an epsilon.
If close enough to face, I assume that face was hit.
I can live with the few false positives, as I shoot over 100M photons each frame anyway.

Bram Stolk

JANUARY 11, 2015 AT 6:05 PM

Oops... that should be 100K photons of course.

★ Tavian Barnes

JANUARY 12, 2015 AT 12:32 PM

Haha I was *really* impressed for a second :)

The cube trick does not require sorting, just selecting the max from three candidates.

Jon olick

JANUARY 22, 2015 AT 2:14 PM

Something to consider here is that 0 * inf =nan which occurs when the ray starts exactly on the edge of a box

---

★ Tavian Barnes

JANUARY 31, 2015 AT 3:10 PM

True, if you want consistent handling of that case while staying branch-free, you have to do a little more work. This is worth another post actually, I'll write one up.

Chris

MARCH 8, 2015 AT 7:10 PM

Thanks for posting this (so long ago)! I used this in my own code, and wondered if it is possible to save 2 of the subtractions by taking advantage of the fact that min(x+a, y+a) = min(x,y)+a (for some constant a):

```
r.offset = (r.n_inv.y * r.x0.y) - (r.n_inv.x * r.x0.x);

bool
intersection(box b, ray r)
{
  double tx1 = b.min.x * r.n_inv.x;
  double tx2 = b.max.x * r.n_inv.x;

  double tmin = min(tx1, tx2) + r.offset;
  double tmax = max(tx1, tx2) + r.offset;

  double ty1 = b.min.y * r.n_inv.y;
  double ty2 = b.max.y * r.n_inv.y;

  tmin = max(tmin, min(ty1, ty2));
  tmax = min(tmax, max(ty1, ty2));
```

```
  return tmax >= tmin;
}
```

★ Tavian Barnes

MARCH 11, 2015 AT 1:15 PM

I think that should work, but there's a couple bugs in your example code. And actually for the test as written you can omit `r.offset` entirely since it doesn't affect the `tmax >= tmin` check at the end. But since you probably want a `tmax >= 0.0` check too, this should work:

```
r.offset = (r.n_inv.y * r.x0.y) + (r.n_inv.x * r.x0.x);

bool
intersection(box b, ray r)
{
  double tx1 = b.min.x * r.n_inv.x;
  double tx2 = b.max.x * r.n_inv.x;

  double tmin = min(tx1, tx2);
  double tmax = max(tx1, tx2);

  double ty1 = b.min.y * r.n_inv.y;
  double ty2 = b.max.y * r.n_inv.y;

  tmin = max(tmin, min(ty1, ty2));
  tmax = min(tmax, max(ty1, ty2));

  return tmax >= max(tmin, r.offset);
}
```

I'll try it out and see how much faster it is, thanks!

Mario

MAY 16, 2015 AT 11:13 AM

What if I want to know the 't' of the intersection?

★ Tavian Barnes

MAY 16, 2015 AT 6:13 PM

$t = t_{\min}$, unless $t_{\min} < 0$, in which case you're inside the box and $t = t_{\max}$.

Cody Bloemhard

MAY 29, 2015 AT 11:11 AM

Very useful! faster the any method i tried. not because of the !(use of divisions), but because the implementation of the slab method is far simpler than others do. They use things(slow things) like

square roots, DOT products and lots of checks with points. now my raycasting method has the speed i wanted. Thanks for that.

Ciyo

DECEMBER 1, 2015 AT 4:39 PM

Hello, many thanks for this useful information!

I have a question about the surface normal. Is there an easy way to get the normal of the intersection point?

Thank you!

Phlimy

SEPTEMBER 6, 2016 AT 11:50 AM

I would really like to know too!

Diego Sinay

MAY 12, 2016 AT 4:30 PM

Hi, great read!
One quick question.
I had no problem using the first implementation for my ray-tracing algorithm, but can't implement that faster version since I can't get the inverse of the direction vector(doesn't it have to be squared?).
Any help is appreciated, Thanks!

★ Tavian Barnes

MAY 16, 2016 AT 12:52 PM

You can just compute (1/x, 1/y, 1/z) as the inverse. You don't have to square it.

Phil

JUNE 23, 2016 AT 2:50 PM

Hey there,

considering I have box with min(1,1,1) and max(2,2,2) and a ray with origin(3,3,3) and direction (2,2,2). Oh. Wait a second. I think I just answered my question. It's ray direction, not target position.

If I want my ray to go from (3,3,3) towards the direction of point (2,2,2) I need a direction like (-1,-1,-1), correct? Oh my, this made me struggle way longer than it should have.

Nevermind me. Thank you for the write-up in both articles. :)

Dave

SEPTEMBER 27, 2016 AT 4:54 PM

Does anyone know why this doesn't seem to work in GLSL for all angles?

INFINITY is defined as `const float INFINITY = 1.0 / 0.0;`

```
float rayAABBIntersection (Ray ray, AABB aabb)
{
	float tx1 = (aabb.min.x - ray.origin.x)*ray.inverseDirection.x;
	float tx2 = (aabb.max.x - ray.origin.x)*ray.inverseDirection.x;

	float tmin = min(tx1, tx2);
	float tmax = max(tx1, tx2);

	float ty1 = (aabb.min.y - ray.origin.y)*ray.inverseDirection.y;
	float ty2 = (aabb.max.y - ray.origin.y)*ray.inverseDirection.y;

	tmin = max(tmin, min(ty1, ty2));
	tmax = min(tmax, max(ty1, ty2));

	float tz1 = (aabb.min.z - ray.origin.z)*ray.inverseDirection.z;
	float tz2 = (aabb.max.z - ray.origin.z)*ray.inverseDirection.z;

	tmin = max(tmin, min(tz1, tz2));
	tmax = min(tmax, max(tz1, tz2));

	if (tmin > tmax)
	{
		return INFINITY;
	}

	if (tmin < 0)
	{
		return tmax;
	}

	return tmin;
}
```

★ Tavian Barnes

SEPTEMBER 29, 2016 AT 9:44 PM

What do you mean "doesn't work"? What happens? Do you have a numerical example that gets evaluated wrong?

Ali

FEBRUARY 1, 2017 AT 9:35 AM

How to find the intersection points of a Cartesian grid and a boundary (e.g. a circle, an ellipse, an arbitrary shape)? I am interested in the intersection of the boundary with the cartesian grid. Any suggestion/algorithm, kindly email me.

Mark R

MAY 16, 2017 AT 11:05 AM

Hello.

Having read both this and part 2, I'm unable to work out what changes I need to make in order to reliably detect intersections when a ray is exactly on the edge of an AABB. I have your implementation here with a simple failing test case:

https://github.com/io7m/raycast-issue-20170516

I've stepped through the code and tried making the changes you suggested in the other article (exchanging Math.max and Math.min for implementations that have the semantics of maxNumber and minNumber) but I can't seem to get it to work.

Maybe I've misunderstood the intent of the other article: What changes do I need to make to reliably catch ray/edge intersections? I care less about efficiency and more about avoiding false negatives.

worst drivers

MAY 17, 2017 AT 11:00 AM

paulao disse:Outra condiÃ§ao seria que eles sedecem os codigos fontes para usar os nossos misseis nacionais e demais aproveitamentos com essa possibilidade.Sem isso eu acho que seria desastrosa uma compra de mirages.Sou PATRIOTA.Sds.