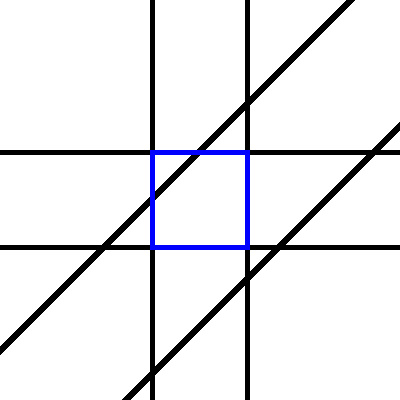
**Fast, Branchless Ray/Bounding Box Intersections**

[May 2, 2011](https://tavianator.com/fast-branchless-raybounding-box-intersections/) [Tavian Barnes](https://tavianator.com/author/tavianator/)

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](http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm). 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.