An Efficient and Robust Ray–Box Intersection Algorithm

Amy Williams   Steve Barrus   R. Keith Morley   Peter Shirley

University of Utah

Abstract

The computational bottleneck in a ray tracer using bounding volume hierarchies is often the ray intersection routine with axis-aligned bounding boxes. We describe a version of this routine that uses IEEE numerical properties to ensure that those tests are both robust and efficient. Sample source code is available online.

Introduction

Naive implementations of ray–box intersection algorithms can have numerical problems for rays that have slopes near zero along any axis. Smits [3] pointed out that properties given in the IEEE floating point standard [1] can be used to avoid explicit tests for these values, but did not provide the implementation details. The following is an implementation of Smits' algorithm. It expects a box with ordered corners `min` and `max`, a ray `r`, and a valid intersection interval of `(t0, t1)` to be given. We assume that the `Vector3` and `Ray` classes are implemented; their usages below should be obvious.

```
class Box {
  public:
    Box(const Vector3 &min, const Vector3 &max) {
      assert(min < max);
      bounds[0] = min;
      bounds[1] = max;
    }
    bool intersect(const Ray &, float t0, float t1) const;
    Vector3 bounds[2];
};

// Smits' method
bool Box::intersect(const Ray &r, float t0, float t1) const {
  float tmin, tmax, tymin, tymax, tzmin, tzmax;
  if (r.direction.x() >= 0) {
       tmin = (bounds[0].x() - r.origin.x()) / r.direction.x();
       tmax = (bounds[1].x() - r.origin.x()) / r.direction.x();
  }
  else {
       tmin = (bounds[1].x() - r.origin.x()) / r.direction.x();
       tmax = (bounds[0].x() - r.origin.x()) / r.direction.x();
  }
  if (r.direction.y() >= 0) {
       tymin = (bounds[0].y() - r.origin.y()) / r.direction.y();
       tymax = (bounds[1].y() - r.origin.y()) / r.direction.y();
  }
  else {
       tymin = (bounds[1].y() - r.origin.y()) / r.direction.y();
       tymax = (bounds[0].y() - r.origin.y()) / r.direction.y();
  }
  if ( (tmin > tymax) || (tymin > tmax) )
       return false;
```

```
    if (tymin > tmin)
          tmin = tymin;
    if (tymax < tmax)
          tmax = tymax;
    if (r.direction.z() >= 0) {
         tzmin = (bounds[0].z() - r.origin.z()) / r.direction.z();
         tzmax = (bounds[1].z() - r.origin.z()) / r.direction.z();
    }
    else {
         tzmin = (bounds[1].z() - r.origin.z()) / r.direction.z();
         tzmax = (bounds[0].z() - r.origin.z()) / r.direction.z();
    }
    if ( (tmin > tzmax) || (tzmin > tmax) )
         return false;
    if (tzmin > tmin)
          tmin = tzmin;
    if (tzmax < tmax)
          tmax = tzmax;
    return ( (tmin < t1) && (tmax > t0) );
  }
```

Note that the reason we check the sign of each component direction is to ensure that the intervals produced are ordered (i.e., so that `tmin <= tmax` is true). This property is assumed throughout the code, and allows us to reason about whether the computed intervals overlap. Note also that since IEEE arithmetic guarantees that a positive number divided by zero is $+\infty$ and a negative number divided by zero is $-\infty$, the code works for vertical and horizontal lines (see [2] for a detailed discussion of this).

Improved Code

The code from the previous section works correctly for almost all values, but there is a problem if `r.direction.x() == -0.0` In this case, the the first `if` statement will be true (`-0 == 0` is true in IEEE floating point), and instead of the resulting interval being $(-\infty, +\infty)$, it will be the degenerate $(+\infty, -\infty)$. The same problem appears when either `r.direction.y()` or `r.direction.z()` are `-0.0`. When such a degenerate interval is obtained, the function will return `false`. The algorithm therefore fails to detect a valid intersection in this situation. While this scenario may seem unlikely, negative zeroes can arise in practice, and indeed have in our applications which is how we discovered this problem. Note how easy it is to generate a negative zero:

```
float u = -2.0;
float v =  0.0;
float w =  u*v;  // w is now negative zero
```

Many implementations of ray–box intersection replace the two divides in each `if` clause with a single divide and two multiplies:

```
divx =  1 / r.direction.x();
tmin = (bounds[0].x() - r.origin.x()) * divx;
tmax = (bounds[1].x() - r.origin.x()) * divx;
```

This is done because the two multiplies are usually faster than the single divide they replace, but it also allows a way out of the negative zero problem. `divx` captures the sign of `r.direction.x()` even when it is zero: `1 / 0.0 =` $+\infty$ and `1 / -0.0 =` $-\infty$. The updated algorithm for the *x* component (*y* and *z* are analogous) is:

```
// Improved method for x component
divx =  1 / r.direction.x();
if (divx >= 0) {
     tmin = (bounds[0].x() - r.origin.x()) * divx;
     tmax = (bounds[1].x() - r.origin.x()) * divx;
}
else {
```

```
      tmin = (bounds[1].x() - r.origin.x()) * divx;
      tmax = (bounds[0].x() - r.origin.x()) * divx;
}
```

Note that it is important to test the sign of `divx` rather than `r.direction.x()` in order for `-0.0` to be properly detected. This does result in an efficiency penalty on some systems because the evaluation of the `if` statement must wait for the result of the divide. Nonetheless, to ensure the correctness of the ray–box test in all cases, this penalty must be accepted. The code with a test on `divx` was first presented by Smits [4]; although he did not explicitly state its advantage for handling zeros, he was probably aware of it because the associated efficiency penalty makes it otherwise unattractive.

Optimizing for Multiple Box Tests

Rays are often tested against numerous boxes in a ray tracer, e.g., when traversing a bounding volume hierarchy. The above algorithm can be optimized by precomputing values that remain constant in each test. Rather than computing `divx = 1 / r.direction.x()` each time a ray is intersected with a box, the ray data structure can compute and store this and other pertinent values. Storing the inverse of each component of the ray direction as well as the boolean value associated with the tests (such as `divx >= 0`) provides significant speed improvements. The new code is fairly simple:

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

// Optimized method
bool Box::intersect(const Ray &r, float t0, float t1) const {
  float tmin, tmax, tymin, tymax, tzmin, tzmax;

  tmin = (bounds[r.sign[0]].x() - r.origin.x()) * r.inv_direction.x();
  tmax = (bounds[1-r.sign[0]].x() - r.origin.x()) * r.inv_direction.x();
  tymin = (bounds[r.sign[1]].y() - r.origin.y()) * r.inv_direction.y();
  tymax = (bounds[1-r.sign[1]].y() - r.origin.y()) * r.inv_direction.y();
  if ( (tmin > tymax) || (tymin > tmax) )
       return false;
  if (tymin > tmin)
        tmin = tymin;
  if (tymax < tmax)
        tmax = tymax;
  tzmin = (bounds[r.sign[2]].z() - r.origin.z()) * r.inv_direction.z();
  tzmax = (bounds[1-r.sign[2]].z() - r.origin.z()) * r.inv_direction.z();
  if ( (tmin > tzmax) || (tzmin > tmax) )
       return false;
  if (tzmin > tmin)
        tmin = tzmin;
  if (tzmax < tmax)
```

```
        tmax = tzmax;
    return ( (tmin < t1) && (tmax > t0) );
  }
```

We ran tests to ensure that the multi-box optimization did not incur a decrease in efficiency for the case in which a single box or shallow bounding volume hierarchy is intersected. Our results show that the optimized method is indeed faster for both cases. While the runtimes are dependent on processor type and scene content, we found these timings to be typical for most scene complexities and architectures.

Scene	Smits' method	Improved method	Optimized method
Single box - 1e8 rays	77.78s	71.39s	66.82s
1e6 triangles in BVH - 1e8 rays	1027.43s	961.23	739.21s

In both the single-box and BVH tests approximately half of the rays fired hit the test object while the other half were near misses. The tests were performed on a Pentium4 1800 MHz processor.

Web Information

Sample C++ source code for the optimized method described above is available online at `http://www.acm.org/jgt/WilliamsEtAl04`.

Acknowledgments

We would like to acknowledge Brandon Mansfield, Steve Parker, and Erik Reinhard for providing test code. John McCorquodale also provided some useful information about the speed of float-point multiplies and divides. Brian Smits' advocacy for BVH intersection methods and care with IEEE properties gave us the initial impetus for this work. The anonymous reviewer of the article provided very helpful comments, and pointed out that the divides could be performed once for a full hierarchy. This work was partially supported by NSF grant 03-06151.

References

[1] IEEE Standards Association. IEEE standard for binary floating-point arithmetic. IEEE Report (New York), 1985. ANSI/IEEE Std 754-1985.

[2] Peter Shirley. *Fundamentals of Computer Graphics*. AK Peters, 2002.

[3] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3(2):1–14, 1998.

[4] Brian Smits. Efficient bounding box intersection. *Ray tracing news*, 15(1), 2002.