



FIGURE 1. Good old terra firma, minus the firma.

Collision Detection Using Ray Casting

TIM SCHROEDER | *Tim is a programmer working at Volition in Champaign, Ill., and a card-carrying member of the Canadian brain drain. When the big Canuck isn't beating his head against a wall trying to solve collision detection problems, he can usually be found doing something sporty. He can be reached at tims@volition-inc.com.*

Game physics can be divided up into three distinct and (usually) separate phases: simulation, collision detection, and collision response, colloquially known as which way do I go, what did I hit, and where do I go from here.

Collision detection is generally the bottleneck of the three. It is a math-intensive task to determine when something hits something else, and even today's high-horsepower PCs and consoles can choke on the load. To throw gas on that fire, games are continually getting more complicated, with more objects to simulate, more polygons to collide against, and more sophisticated physics. As the physics programmer on Volition's recently released Playstation 2 FPS, RED FACTION (RF), I can tell you that flame burns hot.

RF takes physics complexity to a new level, beyond most current games. Volition's proprietary GeoMod engine is designed to allow real-time arbitrary geometry modification, something that hasn't been done in a 3D game before. This means that a player can point a rocket at any wall and create a gaping hole to jump through. The big problem this poses for collision detection is that it eliminates a lot of the preprocessing tricks that can be used to optimize calculations. A BSP tree, for example, is not something that you can recompute between frames if the level geometry changes.

A second design challenge for RF is a very ambitious physics engine. When the player blows up a bridge (for example), big chunks of rock should break off and tumble to the ground in a realistic way, maintaining angular momentum and picking up spin from friction with the ground. These computations take up a lot of CPU cycles and require a pretty accurate collision detection system to look right.

So, we can use no preprocessing and we need high accuracy. What kind of collision detection is going to work with these terrifying caveats? Let's take a brief look at some of the options.



Intersection Testing

By far the most popular form of collision detection in games, intersection testing is conceptually simple and robust. It works by checking an object's desired position and determining whether it will intersect the world or another object. If it will intersect, the object is backed up and tested again, recursing until no collision occurs. The common algorithm for this process is a binary subdivision of the distance between the last valid position and the desired position of the object. The beauty of binary subdivision is that it allows you to find a happy medium between accuracy and speed by tuning the number of iterations.

There are a number of methods for doing the actual intersection test, from axis-aligned bounding boxes to convex hulls employing Voronoi regions. There are also a number of preprocessing steps that can make the intersection testing faster. A common method is to generate a BSP tree of the world geometry. This simplifies the intersection test to a series of plane equations and nicely subdivides the world into a hierarchical tree of valid areas. You can also use a simplified set of planes to define the boundaries of the world separate from the rendered geometry to cut down on the complexity of the collision detection.

The main problem with intersection testing is that for any moving object you can choose a sufficiently large frame time and a sufficiently high velocity such that the object moves completely from one side of an obstacle to the other without intersecting it. Thin walls pose a significant problem for that reason. This was a big strike against that method when evaluating schemes for RF, since the player can make all manner of malformed geometry by blowing away chunks of the world. In a controlled environment restrictions can be put on level designers to solve this problem, but it's impossible to do the same for the player.

Ray Casting

Ray casting is fundamentally different from intersection testing in that it projects the object along the path from its last valid position to its desired position and attempts to determine the exact time that the object collides with an obstruction (see Figure 2). This is a fairly simple operation with a particle, but it becomes much more expensive with objects of volume. To cut down on complexity, the object is generally approximated with simple

shapes, such as a bounding box, a collection of spheres, or possibly a convex hull.

The advantage of this system is that it only requires one iteration and the results are more accurate than multiple iterations of binary subdivision intersection. The drawback is that the collision calculation is more expensive than an intersection test for objects of equal complexity. The trade-off between speed and accuracy becomes more apparent at high velocities.

The preprocessing steps that are applicable to intersection testing are just as applicable to ray casting. Unfortunately, most of them assume a static world and thus are not applicable to RF.

Our Design

After evaluating the options for RF, ray casting came out the clear winner. Even so, there was a whole host of optimizations we needed in order to make the collision detection speedy enough. In the rest of this article, I'll detail the collision detection methods implemented in RF. But first, let's look at a roadmap of the system as a whole.

At the highest level, the world in RF is divided into a series of rooms separated by portals. This is a common rendering optimization that nicely breaks the world up into bite-sized chunks. The polygons located in a room are organized into a binary tree of bounding boxes. At the lowest level, the polygons themselves each have individual bounding boxes around them. In order to collide an object with a polygon, the object's bounding sphere is used. The object first has to pass through the right room, then the right section of the bounding box tree, and finally the bounding box of the polygon.

Now that you have an idea of how our collision detection system works, let's start with the basics and work our way up.

Simple Particle/Polygon Collisions

The simplest form of ray casting boils down to a particle (which has no volume) moving through space, and a polygon. There are two tests to perform when doing collision detection with a particle:

1. Where does the particle cross the plane of the polygon?
2. Is that collision point inside the polygon?

The first test is greatly simplified if the plane equation of the polygon is known. In RF each world face stores its plane equation to keep from having to recompute it every time it is needed.

Let me clarify what constitutes a collision. A collision occurs when an object passes from the front of a polygon to the back. Objects moving in the other direction are not considered.

Now that we're clear, let's collide our particle with the plane. To do this we need the initial position of the particle, its path during this frame, and the plane equation of the polygon. The algorithm is very simple:

1. Find the distance (**dist**) from the particle to the plane.
2. Find the length (**len**) of the path vector along the normal of the plane.
3. Determine the time of the collision using distance divided by length.

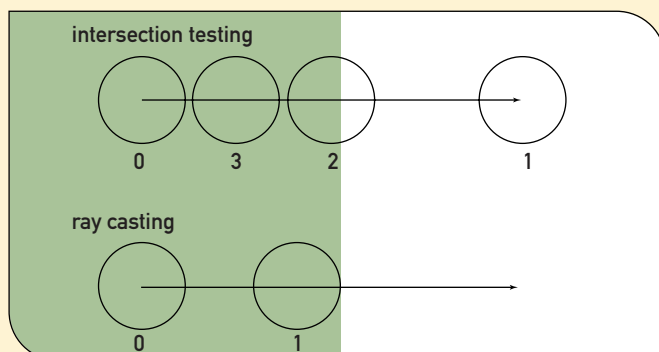


FIGURE 2. Intersection testing versus ray casting.

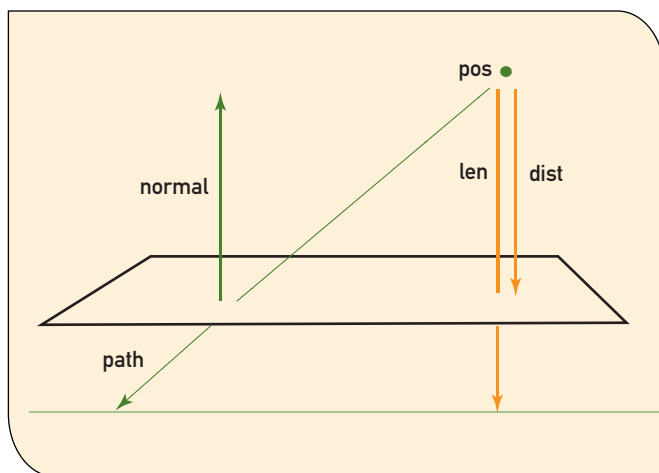
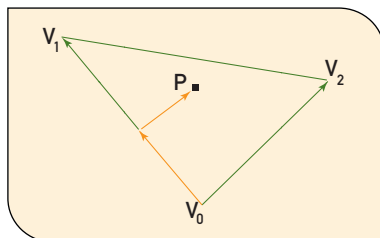


FIGURE 3 (above). Colliding with the plane.
FIGURE 4 (right). Determining if the hit point is inside the polygon.



If the time of the collision is negative, the plane is behind the particle. If the time is greater than 1, the plane is too far away to hit this frame. The code to find the collision point follows. Note that *path* is the vector from the current valid position of the particle to the desired position, and it is not normalized.

```
bool collide_line_plane(vector &pos, vector &path, plane &pl,
                      float &time)
```

```
{
    float dist, len;

    dist = pl.distance_to_point(pos);
    if( dist < 0 ) {
        // behind plane
        return false;
    }

    len = -(path * pl.normal); // Vector dot product
    if( len < dist ) {
        // moving away from plane or point too far away
        return false;
    }

    time = dist/len;
    return true;
}
```

If the particle has crossed the plane this frame, the time variable will contain the fraction of the frame time at which the collision occurred. The collision point *P* can then be found with the following parametric equation:

$$P = (\text{path} \times \text{time}) + \text{pos}$$

Now we need to see whether that hit point is within the boundaries of the polygon. To do that, we reduce the problem to two dimensions by eliminating one of the world axes from our calculations. We eliminate the dominant component of the normal to get the best projection onto a plane formed by two of the three axes. Representing the vectors as arrays, those two remaining axes are identified by the indices *j* and *k*. We then compute 2D vectors for the hit point *P* and two vertices *V*₁ and *V*₂ relative to the third vertex, *V*₀:

$$\begin{aligned} A &= V_1 - V_0 \\ B &= V_2 - V_0 \\ C &= P - V_0 \end{aligned}$$

This allows us to describe the hit point as the sum of two of the edges of the polygon:

$$C = \alpha A + \beta B$$

We can now solve for α and β using determinants:

$$\begin{aligned} \alpha &= (B_k C_j - B_j C_k) / (A_k B_j - A_j B_k) \\ \beta &= (A_k C_j - A_j C_k) / (A_j B_k - A_k B_j) \end{aligned}$$

Three conditions must be satisfied for the point to be inside the polygon:

1. $\alpha \geq 0$
2. $\beta \geq 0$
3. $\alpha + \beta \leq 1$

If all three conditions are true, we have found a collision. Note that this algorithm only works on triangles. *N*-sided polygons will have to be triangulated first or the algorithm modified to loop through the vertices of the polygon three at a time. For further detail on this algorithm, see For More Information.

Crank up the Volume

Colliding rays with polygons is all well and good for a particle system, but our objects have some volume, so we need something more complex. There are a number of possible tech-

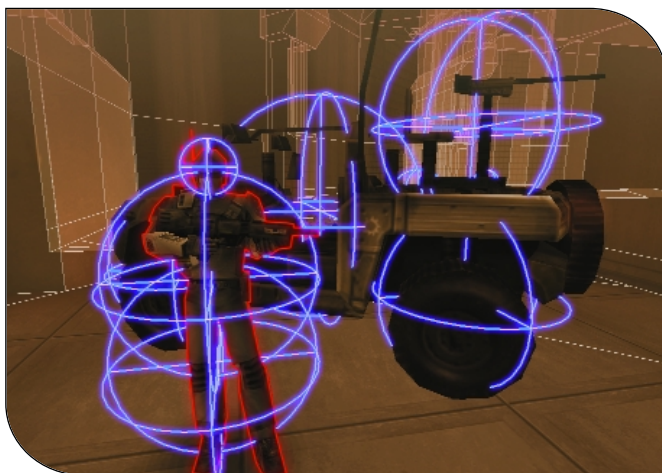


FIGURE 5. Objects are just collections of spheres.



niques for approximating an arbitrary object to use in collision detection. These include bounding boxes, collections of linked particles, and convex hulls. In RF, objects are approximated by a collection of spheres.

Spheres have a number of wonderful properties for collision detection, not the least of which is that a sphere has the same shape from every viewing angle. This makes colliding a sphere with a polygon very similar to colliding a particle. When a sphere bumps into a plane, it's always the point closest to the plane that touches, so if we find that point we can then pass it and the sphere's path into the `collide_line_plane()` function described earlier to determine the hit point. The point on a sphere (centered at `pos`) that's closest to the plane `pl` is found using the equation:

$$P = \text{pos} - (\text{pl.normal} \times r)$$

Sphere/edge collision. The complication, however, is that if the hit point isn't located within the polygon, it is still possible for the sphere to have hit just an edge of the polygon, or even just a vertex. Let's look at the edge case first — we need a function that can determine the collision of a sphere with a line segment (an edge of the triangle).

Two conditions are true when a collision occurs: First, the distance from the line to the sphere center is equal to the radius of the sphere, and second, the vector from the hit point to the sphere center is perpendicular to the line (in other words, their dot product equals 0). If we put our conditions into equations we get:

1. $(\text{hit_pos} - \text{hit_point}) \cdot (\text{hit_pos} - \text{hit_point}) = r^2$
2. $(\text{hit_pos} - \text{hit_point}) \cdot (V_1 - V_0) = 0$

Where the position of the sphere at the time of impact is given by the parametric equation:

$$\text{hit_pos} = (\text{path} \times \text{hit_time}) + \text{pos}$$

and the hit point on the edge is likewise given by the parametric equation:

$$\text{hit_point} = [(V_1 - V_0) \times \text{dist}] + V_0$$

The two things we need to compute are `hit_time`, which is the time when the sphere hits the line, and `dist`, which is the distance from `V0` along the line to where the sphere touches when they collide. `hit_time` must be between 0 and 1 for a collision to occur in this frame, and `dist` must be between 0 and 1 for the hit point to be between the end points of the triangle edge. That leaves us with two equations and two unknowns. If you're like me this is when you break out Mathematica.

Once the equations have been simplified and solved, we wind up with a quadratic function in the form $y = ax^2 + bx + c$. The roots of that function are the `hit_time` values for collisions of the sphere with the line subject to our constraints. The coefficients (a , b , and c) of the quadratic turn out to be an unholy concoction of dot products:

$$\begin{aligned} a &= (V_e \cdot V_s)^2 - (V_e \cdot V_e)V_s^2 \\ b &= 2 \times [(V_d \cdot V_e) \times (V_e \cdot V_s) - (V_d \cdot V_s) \times (V_e \cdot V_e)] \\ c &= (V_d \cdot V_e) \times (V_d \cdot V_s) + [r^2 \times (V_e \cdot V_e)] - [(V_d \cdot V_d) \times (V_e \cdot V_e)] \end{aligned}$$

Once we have a , b , and c we can plug them into the quadratic formula and see what comes out. The quadratic formula, which you probably thought would never come in handy outside a high school math class, is:

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

If $b^2 - 4ac$ (the determinant) is less than 0, the solution is composed of imaginary numbers, which means that the sphere never got close enough to the line to collide. If the determinant equals 0, there is only one root, which means that the sphere grazed the line tangentially. There is typically no collision response necessary for that case, so we can safely ignore it as well.

If we do get two roots, the smaller is the one that interests us because it is the hit time of the collision. The larger represents when the backside of the sphere would have collided with the line segment on its way past.

Sphere/vertex collision. Unfortunately we're still not done. If the sphere didn't hit the line within the vertices, it could still hit one of its end points, so we need to test for collision with them as well. The good news is that since we test every edge of the polygon, we only need to check the first vertex on each edge. Also, some of the work we did to set up for the edge collision is applicable to the vertex collision.

$$\begin{aligned} a &= V_s \cdot V_s \\ b &= 2 \times (V_d \cdot V_s) \\ c &= (V_d \cdot V_d) - r^2 \end{aligned}$$

All that's left is to plug the new a , b , and c into the quadratic

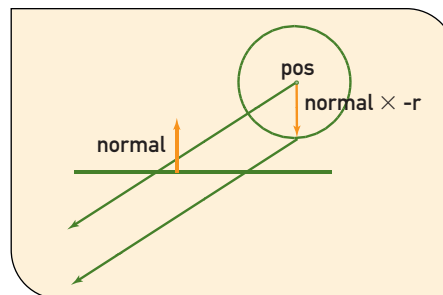
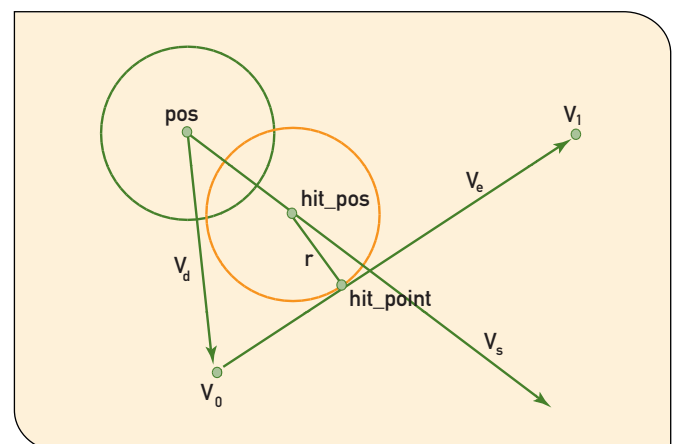


FIGURE 6 (left). Turning a sphere collision into a particle collision. FIGURE 7 (below). Sphere/line segment collision.



formula and find the roots the same way we did for the line.

One thing to note is that sphere/line segment collision tests are by no means cheap, yet they potentially occur three times for every polygon. Further, each line segment in the scene could be tested twice, since polygons share edges. One thing that we did on RF that eased this pain was to check the edge against a bounding box around the sphere's path. The line segment/bounding box test quickly culls out the edges that couldn't possibly collide and makes a nice segue to my next point.

Putting Polygons into Boxes

Now we have the tools for throwing spheres around and colliding them with the world. Unfortunately, unless your world consists of just a few boxes, this isn't going to be fast enough to run in real time. The big problem is that planes are infinite, so if you don't have some notion of locality, you are forced to do expensive collision tests with every polygon in the entire world. The collision detection system needs a fast way to cull the number of faces it has to check. Enter axis-aligned bounding boxes (AABBs).

The simplest collision test you can perform is the intersection of two AABBs, which takes at most six comparisons. If each world face has a bounding box around it, and we create a bounding box around the path of the sphere, then we can quickly and easily tell whether that object can possibly hit the polygon. In fact, it is usually faster to compute these face bounding

boxes on the fly and use them for culling than it is to do the expensive sphere/face tests.

A problem with the bounding box test is that if the sphere moves quickly its bounding box will be quite large, particularly if it moves diagonally to the world axes. A slightly more expensive but potentially much more accurate test is to intersect the sphere's path vector directly with the face bounding box. But when colliding the path of a sphere with the bounding box, the radius of the sphere must be added to the bounding box for this to work properly. The line segment/bounding box intersection function we used in RF eliminates twice as many faces as the standard bounding box intersection test and is only slightly slower.

The way this works is similar to Cohen and Sutherland's 2D line-clipping algorithm (see For More Information). Each end-point of the line segment is given an outcode based on which sector it is in. The trivially true case, where either of the endpoints is inside the box, and the trivially false case, where both endpoints are on the same side of the box, are found quickly with minimal effort. If further work is required, the outcodes isolate which planes to clip the line segment against.

For each plane of the bounding box that clips the line segment we need to check the intercept point on the plane to see whether it actually falls on the box. This is accomplished by determining the distance along the line segment at which it intercepts that plane. If we are clipping to the bounding box's maximum x plane, for example, this equation is:

$$\text{dist} = (V_{1x} - V_{0x}) / (x_{\text{max}} - V_{0x})$$

And the intercept point would be:

$$\text{intercept} = [(V_1 - V_0) \times \text{dist}] + V_0$$

In our example, the y and z values for the intercept are then checked to see whether they fall within the minimum and maximum values for the bounding box.

Note that any line segment that intersects the box will cross at least two planes, but finding one is sufficient to know that we can't cull this polygon. The time and position of the intersection can be ignored if the bounding box test is used only for culling.

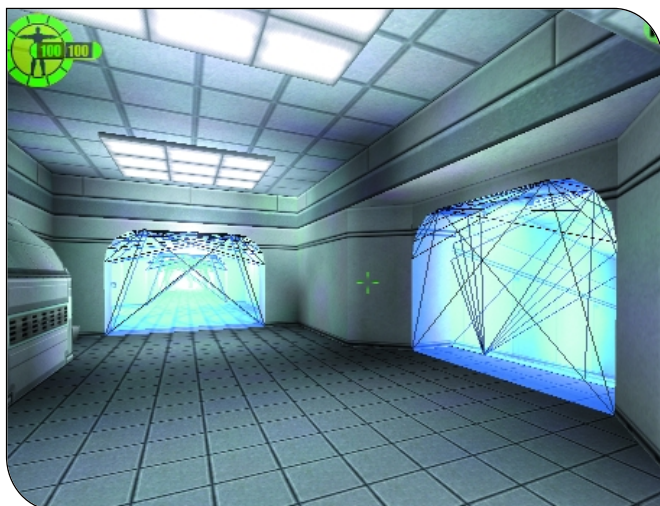
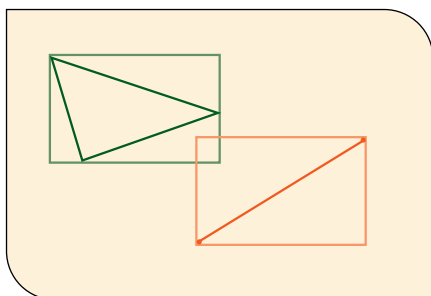
Putting Boxes into Bins

Instead of thousands of expensive sphere/polygon checks, our engine now does thousands of cheap bounding box checks. That's a big win, but it's still a lot of number crunching. We need a way to cull out more than one polygon at a time so that we don't need to look at every polygon in the level for every potential collision. We'll use some spatial partitioning to narrow down the list of potential collision candidates.

A very simple but effective partitioning system is to create a binary tree of AABB "bins" which hold the polygon AABBs. Each bin contains either two smaller bins or a collection of polygons. Creating the tree is easy, and it can be tuned for performance. Once the tree has been created, bounding box testing becomes a recursive procedure of checking whether an object is within successive child boxes. The creation algorithm is:

1. Create a bin that encompasses all polygon bounding boxes.

FIGURE 8 (right).
Bounding box
inaccuracy.
FIGURE 9 (below).
Rooms separated by
portals.



2. Divide the bin in half across its longest axis.
3. Use the center of each polygon to determine which bin it belongs in.
4. Add the polygon AABB to the proper split bin.
5. Repeat step 2 for both new bins.

The recursion can be truncated as necessary with a limit on the depth of the tree, a target for the number of polygons in a bin, or a maximum bin volume. The ability to tune the algorithm with these parameters was vital for RF, because the tree has to be recalculated whenever the level geometry changes.

One undesirable property of the resultant tree is that there can be a fair amount of bin overlap if the polygons are large. This means that an object can be found in both child bins at the same time, which defeats the purpose of having the tree in the first place. In practice, however, using the tree drastically cuts the number of AABB checks performed, and the method scales up very well with level size and polygon count.

I mentioned earlier that the RF renderer is room/portal based. It makes sense to utilize that natural partitioning to help out the collision detection. Since all polygons in a level are explicitly linked to exactly one room, it is a simple task to create a bounding box tree for each room. This is also a big win when recalculating a tree after a geometry modification occurs, since only the affected room needs to be recalculated.

Floating-Point Fun

One thing left to mention is how to deal with numerical imprecision. If computers could represent real numbers exactly, there would be no problem implementing mathematical formulae in code. In practice, however, computers are finite and imprecise. Colliding a sphere with a plane when the two are almost touching, for example, is not guaranteed to work without error. Because of this imprecision it is a good idea to incorporate some judicious fudging into the equations. It is in the fudge factors that the art of collision detection is found.

The best place I've found to add fudge is to the hit time returned from the collision detection functions. If you collide a sphere with the world and you determine that it hit a wall 60 percent of the way into the frame, you can artificially manipulate that hit time to make your sphere stop short of the wall. You don't want to do that based strictly on time, however, since slow objects would then stop closer to walls than fast ones. A better option is to make this buffer distance-based, and more than that, to maintain the distance from the polygon rather than the hit point.

To adjust the hit time, first find the distance from the current position to the plane of the polygon. Then subtract the buffer distance from that and recalculate the percentage along the original path.

```
len = - (path • normal)
dist = (len × hit_time) - buffer_dist
new_hit_time = dist / len
```

It's important to keep the adjusted hit time from going negative, even though that means that the object is inside the buffer

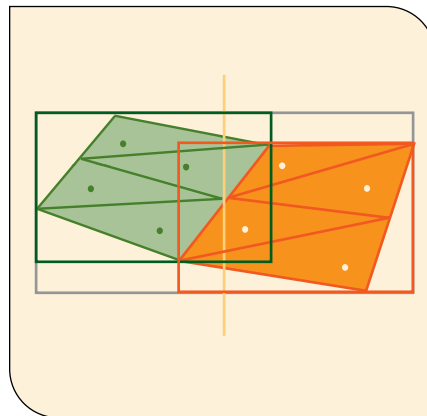



FIGURE 10. Clipping a line segment to an AABB.

zone. A negative hit time will back the object up and might put it through another polygon behind it, which is much worse.

Summing Up

Like most things, ray casting has advantages and disadvantages. It works very well within the framework of RED FACTION's unique challenges, but it isn't the solution to every game's collision detection needs. However, even if ray casting doesn't float your particular boat, the optimization techniques used in RED FACTION can be incorporated into other collision detection systems.

Ray casting is a well-researched area of computer graphics, and there are some great resources available on the web. The most useful one I've found is The 3D Object Intersection page (see For More Information), which provides links to various intersection algorithms and source code repositories in a handy grid. Another good resource is David Andsager, but he's generally only available at the Volition offices. The source code for the techniques detailed in this article is available for download from *Game Developer's* web site at www.gdmag.com. 

FOR MORE INFORMATION

RAY-POLYGON INTERSECTION ALGORITHM

Badouel, Didier. "An Efficient Ray-Polygon Intersection." In *Graphics Gems*, edited by Andrew S. Glassner, pp. 390–93. San Diego: Academic Press, 1990.

THE COHEN-SUTHERLAND LINE-CLIPPING ALGORITHM

Foley, James D., and others. *Computer Graphics: Principles and Practice*, 2nd ed. Reading, Mass.: Addison-Wesley, 1996. pp. 113–17.

THE 3D OBJECT INTERSECTION PAGE

www.realtimerendering.com/int