Practical Collision Detection

Game Developers Conference 1997 Lecture #177
Jonathan Blow jon@number-none.com http://number-none.com

Target Audience

Programmers of realtime 3D applications who intend to implement sophisticated collision detection schemes.

Abstract

Most modern 3D games require some form of collision detection. This paper presents some basic brute force methods of collision detection between polyhedra, successively refining the techniques until we are left with a system that runs quickly. An implementation is examined in which a game engine handles 32 players and hundreds of objects in realtime.

Modus Operandi

Over the past year, we have been experimenting with methods of collision detection for use in games. The goal of this paper is to communicate the experience we have gained. Being an exposition of "practice and experience", this paper does not present any *new work* in that it does not plant a new condominium tract amid the burgeoning fields of computer science. We hope merely to provide observations and ideas that will be useful to others exploring this subject. All the ideas presented here (and many more) would be evident to anyone who sat down and experimented with collision detection for a time. Our main goal is to save you some of that time.

This paper is constructed in a form that matches the evolution of our collision detection system over time. In the first section we begin slowly, taking small steps until a foundation is firmly established. We detail the history of our progress, highlighting key discoveries. Finally, we speed through some advanced concepts, and leave off with some jaded summaries of the current state of collision detection research.

I. What Do You Want?

We wanted to make a game with "good collision detection", but we didn't take much time to clarify to ourselves what precisely that meant. We knew that we wanted objects to have the shapes of arbitrary polyhedra, and we knew that we didn't want grossly inaccurate collision results: that is, we never wanted to see objects bounce off empty space, and neither did we want to see them interpenetrate.

A First Try

If two objects are sticking through each other, and they are both closed polyhedra, then at least one face of one object will be penetrating at least one face of the other object. If we can detect this case as soon as it occurs, then "fix" the situation by moving the objects slightly so that they no longer penetrate, then we should be done.

Therefore we decided that the logic governing object motion would proceed in a loop like this:

```
while the game is running
        foreach entity in the world
                update(entity)

update(entity):
        entity.old_position := entity.current_position
        modify entity.current_position based on entity.velocity and other factors
        if Colliding(entity) then entity.current_position := entity.old_position

Colliding(current_entity) → bool:
        foreach entity in the world
                if entity != current_entity then
                        if Entities_Collide(current_entity, entity) then return true
        return false

Entities_Collide(e1, e2) → bool:
        foreach polygon p1 in e1
                foreach polygon p2 in e2
                        if polygons_intersect(p1, p2) then return true
                        return false
```

We have not defined 'polygons_intersect' here. Descriptions of how to determine the intersections of two polygons can be found in [O94]. It is easier and faster to determine intersections of convex polygons than nonconvex ones, so one may wish to compose one's models of convex polygons (there are many good reasons to do this, most of which have to do with graphics.)

When objects collided, we chose to move them back to their starting positions, because if those were "safe" spots for the objects a moment ago, they will probably still be safe. One alternative is to search through space to find new positions for the objects, which is icky. But there are complications in the method that we chose: for example, when an object is moved back to its starting point, we must make sure that it is really still safe; if not (because an object has moved into that space in the meantime), we must move the offending object back to its own starting position. The worse that can happen is that we have to revert every object in the world to its starting point.

For the most part, though, the pseudocode above illustrates an extremely basic, extremely simple, and extremely *slow* way of doing collision detection. Why is it so slow? If we have two

objects, each of which is composed of 100 polygons, then when they collide, that loop body in Entities_Collide will be executed a *lot* -- 100^2 times in most cases. Finding whether two polygons intersect is not the fastest operation known to man. On current-day computers, doing it 10,000 times is far from instantaneous. And doing it 10,000 times for each pair of objects that may collide, updating the world many times per second, becomes impossible.

It's dreadfully slow, but it *works* (with a few exceptions, which we'll talk about later). And something that works is not a bad starting point.

The First Filters

One way to speed up a slow algorithm is to install "filters" which keep the slow part from getting run when it doesn't need to, or which can come up with an easy answer using inexpensive techniques. Immediately we put into place two simple filters that most experienced programmers would take as givens:

A) Segregation is Good

We divide the world into a regular grid that partitions objects into smaller groups, with the goal of reducing the number of entity-entity comparisons the system needs to make. Every time we move an object, we compute which squares of the grid it overlaps. For our engine we chose a two-dimensional grid, although it is a true 3D world, because the game is played over a landscape and we do not expect many objects to be piled on top of each other.

B) Bounding Spheres

Before doing the polyhedron-polyhedron check, we look at the distance between the centers of the two objects. If they are further than the sum of the two objects' radii, the objects cannot

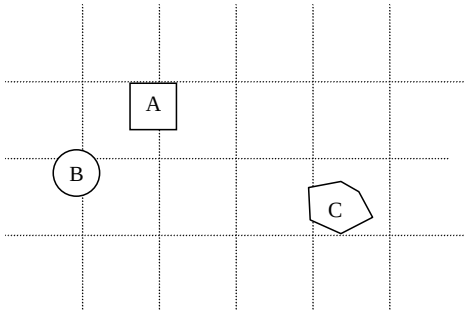

Figure 1: Because entities A and B overlap into a common grid square, they will be collision-tested against each other; C will be tested against no-one.

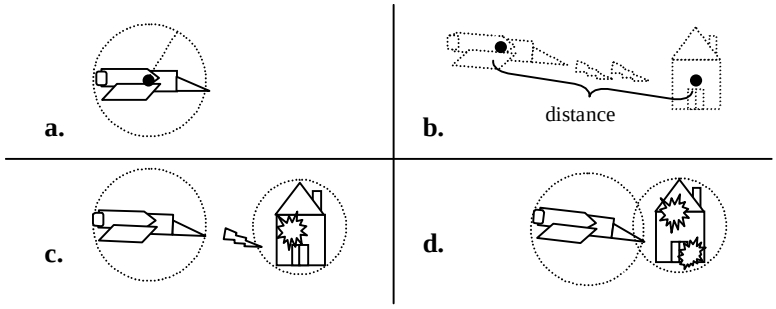

Figure 2: a: A game entity, its position (center point), its bounding sphere and bounding radius. **b:** The distance between two objects' center points. **c:** The two objects are distant enough that their bounding spheres do not intersect; therefore, they cannot collide. **d:** Now the bounding spheres intersect and, if we're lucky, the objects will collide soon!

Distance **D** between P and Q is: $\sqrt{(q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2}$ $\quad D^2 = (q_x - p_x)^2 + (q_y - p_y)^2 + (q_z - p_z)^2$

Bounding spheres intersect if $(D^2 <= (r_p + r_q)^2)$, where r_p and r_q are the radii of the bounding spheres.

Figure 3: Some math.

possibly interpenetrate. To speed this up slightly, we can check the distance squared versus the sum of the radii squared (this avoids a costly square root operation).

C) I Prefer Jersey

Actually, we originally had a third filter that ran before the bounding sphere test, which looked at the Manhattan distance between the objects' centers. This was not really worth the bother (It almost never saved any execution time worth worrying about) so we deleted it.

Filling In the Gaps

The code discussed so far tests for interpenetration, but that might not be adequate. I've never seen a realtime object simulator in which movement wasn't discrete; that is to say, motion occurs by teleporting objects from place to place. The illusion of smooth motion arises because the distances by which the objects "jump" are very small. But the faster an object is moving, the further it must jump during a fixed timestep. If an object jumps far enough during one update, it could "miss" another object, appearing to fly right through it (or part of one object could miss part of another, causing strange things to happen). Our quick fix for this was to create a "speedbox" around the object; the speedbox was a bounding box that enclosed the full volume of space through which the object might pass during one update. When testing for collisions we would use the speedbox's shape instead of the object's actual shape. This violated our original concern that collisions should be very accurate, but we waved our hands and said that since the object was moving so quickly, nobody would see what was going on anyway.

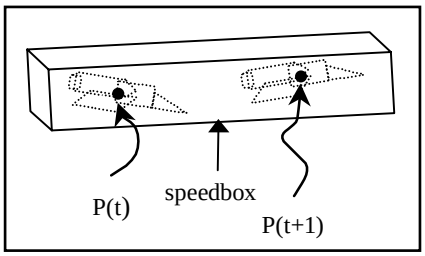

Figure 4: When an object moves too quickly, we replace it with a conservative bounding volume.

See, games are cool because you can wimp out like that, any time you want. The most drastic (and only theoretically sound) alternative we know of is to represent shapes mathematically as functions of their initial space occupancies, their velocities, and time, and then solve huge sets of simultaneous equations to determine collision. We do not expect this approach to be computationally feasible any time this millennium.

Divisiveness is Good

It's an ancient fact that, if two convex polyhedra do not intersect, one can always find a dividing plane between them [Rab95]. Exploiting this fact seemed like a good idea. We didn't want to limit our objects to convex polyhedra, or have to express them as being composed of such. But it is still true that, if objects are *mostly* convex, then you can *usually* find a dividing plane between them (illustrated in figures 5-7).

So we wrote another filter called "plane divides entities", which would heuristically try to find a dividing plane between two objects. And we saw that it was good.

By this point we had built up a stack of “easy”-to-compute filters that would happen before the final collision detection, which became known as the “hard case”. But the hard case was, so to

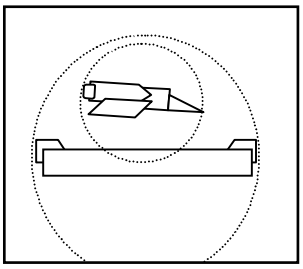

Figure 5, Tragedy of the Spheres: Bounding spheres are woefully inadequate for some common gameplay situations, such as the landing depicted here.

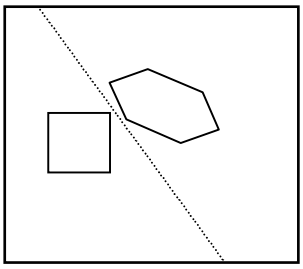

Figure 6: Convex polygons. Dividing line. You can’t find two nonpenetrating convex polygons between which there is no dividing line. Come on, try. I dare you. The same principle applies to polyhedra in 3-space.

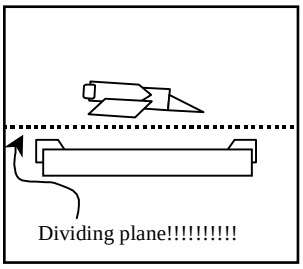

Figure 7: Here we find an easy dividing plane between the two objects.

speak, still too hard. And though the filters were very helpful, there were cases when they just failed to apply. When two objects got very close to each other in ways that left no easily-found dividing plane between them, the game slowed down unacceptably. We knew about BSP trees, so we cracked open some references and began writing some new hard-case code.

We will not attempt to explain BSP trees fully in this paper. Instead, we refer the reader to [Chin95], [FV90], and [Wade95], and provide a brief explanation for the sake of context.

BSP trees consist of a hierarchy of planes, where each plane divides a region of space into two halfspaces. We can use them to describe a solid object by adopting the convention that each separating plane of a leaf node describes a portion of the object’s surface, where one of the plane’s halfspaces represents the inside of the object, and the other represents space that is outside.

Binary tree organizations of n nodes can often allow search operations to complete in $O(\log(n))$ time, and the collision detection we are trying to perform is fundamentally a search. Intuitively, if we use BSP trees well when testing two objects for intersection, we should be able to handle a “hard case” collision test in something like $O(\log(n)^2)$ time, where n is a typical number of polygons contained by an entity.

We can employ BSP trees to speed up collision detection by using their spatial-partitioning properties to reject polygons early. If we are detecting a collision between two entities A and B, and if we know that A lies entirely on one side of some plane P that cuts through B, then we need only test A against the parts of B that are on the same side of the plane as A. (In a sense, this is a more sophisticated version of the Plane Divides Entities test, where the dividing plane eliminates only part of an object, rather than the whole thing.) BSP trees provide us with a myriad of such planes. So if we recurse down the BSP tree of B, finding whether A’s bounding sphere intersects each separating plane (a very cheap operation in itself), we can ignore many polygons of B that have no chance of colliding with A. For each polygon of B that passes this filter, we will call a procedure that compares it with every polygon in A. Some sample C++ source code to do this is presented below:

```
struct Polyhedron {
    List *faces;        // All polygons contained in this object
    Point center;       // Center of the object
    float radius;       // Radius of its bounding sphere
    BSP_Node *bsp_tree; // BSP tree representing this object
};

enum flag_value {
    TOUCHES_POS_HALFSPACE = 0x1,
    TOUCHES_NEG_HALFSPACE = 0x2,
    SLICED = 0x3        // Touches both halfspaces
};

struct BSP_Node {
    float a, b, c, d; // Coefficients of plane equation (ax + by + cz + d = 0)
    List *polygons;   // Polygons that are coplanar with said plane
    BSP_Node *positive, *negative;  // Contents of each halfspace
};

bool object_hits_world(BSP_Node *node, Polyhedron *solid) {
    if (node == NULL) return false;
    int status = classify(node, solid->center, solid->radius);
    if (status == SLICED) {
        if (test_solid_against_polygons(node->polygons, solid))
            return true;
    }

    if (status & TOUCHES_NEG_HALFSPACE) {
        if (object_hits_world(node->negative, solid)) return true;
    }

    if (status & TOUCHES_POS_HALFSPACE) {
        if (object_hits_world(node->positive, solid)) return true;
    }

    return false;
}

int classify(BSP_Node *node, Point center, float radius) {
    float distance = (center.x * node->a) + (center.y * node->b)
                   + (center.z * node->c) + node->d;

    int status = 0;
    if (distance - radius <= 0.0) status |= TOUCHES_NEG_HALFSPACE;
    if (distance + radius >= 0.0) status |= TOUCHES_POS_HALFSPACE;

    return status;
}

bool test_solid_against_polygons(List *polygons, Polyhedron *solid) {
    Polygon *face1, *face2;
    Foreach(polygons, face1, {
        Foreach(solid->faces, face2, {
            if (polygons_intersect(face1, face2)) return true;
        });
```

```
    });

    return false;
}
```

We can go further than this: after selecting potentially colliding polygons from B, rather than testing them against all of A, we can drop them down A's BSP tree, eliminating collision tests with much of A. By the time we're done with all that, we should end up performing relatively few polygon-polygon intersection tests.

In our implementation, we choose the smaller of two potentially colliding objects as A, and the larger as B (judging by the radii of their bounding spheres). The reasoning behind this was that the smaller object was more likely to fit between the larger's partitioning planes, thus reducing the number of polygons considered (see figure 8).

To facilitate the testing of individual polygons from B against A's BSP tree, we chose to store a center point and bounding radius on each polygon of each object model. This is a controversial choice as it increases memory usage, and as a bounding sphere is a fairly pessimistic bounding volume for a polygon.

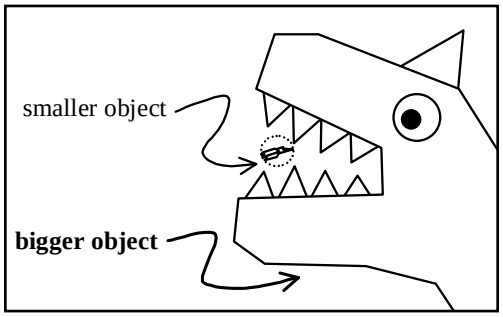

Figure 8: Smaller object fits nicely between the BSP planes defined by Friendly Sock Monster's teeth.

Early Hit Detection

By now things were a lot faster than what we'd started with, but of course we still wanted to make them faster. We figured that, say, if you're moving toward a wall and you collide with it, and you're going at a reasonable speed, you won't jump too far through the wall — chances are that some vertices from your object will actually end up inside the wall (this is illustrated in figure 9.)

So we added a new test that classified the necessary vertices of each object into the other object; if a vertex from A, for example, ends up on the interior of B, you know that they have collided, without performing any polygon intersection tests. Though this test is certainly not sufficient to determine a collision (again, see the figure), it is faster than comparing polygons.

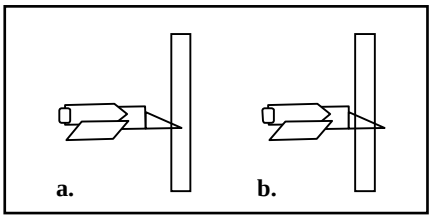

Figure 9: Under favorable conditions, **a** is more likely to occur than **b**.

Eventually we took this test back out. It did speed up the detection of a hit between two objects, but this had a negligible effect on the speed of the game as a whole, for one simple reason: objects almost never collide with each other. For example, suppose you throw a big rock at your friend's face. The rock will travel toward your friend for some time, with little computation being performed because of culling by the early collision filters. But there will come a time when the rock is very close to your friend, and if

he's flinging out his arms in desperation to (unsuccessfully) shield himself from the rock, it will be difficult to find a simple dividing plane between your friend and the rock. Therefore, there will be many update cycles during which the rock is close to your friend, but hasn't hit him yet. After the rock hits your friend's head (if he's not a total klutz he'll have at least managed to turn his face away so that he will not require much plastic surgery), the rock will bounce off and travel back in something like the opposite direction. As you can see, there will be many update cycles during which the rock is close to your friend but not hitting, and only one during which it hits (figure 10).

If the "miss" cycles are much more expensive than the "hit" cycle, it doesn't matter how much faster the "hit" test becomes. This is a basic principle of optimization that we failed initially to think about, and so perhaps we deserve a few rocks ourselves.

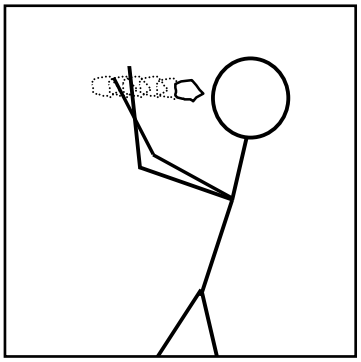

Figure 10: The projectile spends much time close to the target but almost no time actually interpenetrating. You can empirically observe this yourself; all you need is a good friend (hard to come by) and a hefty yet very sharp rock (abundant).

Aha!

But then we did do something that sped up the general case, which was to perform all those BSP tests substituting a bounding box shape for A instead of its true shape. Only if some parts of B collide with A's bounding box do we go and do the full test against A. This does not take too much computation (relatively), and it speeds up the great majority of the near-miss cases that occur in our particular game.

Finally, we added a simple "bounding box safety" test that ran before Plane Divides Entities, which quickly checked to see whether the two objects' bounding boxes overlapped. (This test is similar to the OBB testing discussed later in this paper, but with only one bounding box per object.)

Accuracy is a Virtue

We have presented a few ways of using bounding volumes and dividing planes, but when using these one must be very careful not to be bitten by the evil spectre of Numerical Roundoff Error. To illustrate this, we'll look at the bounding sphere test. We might initially compute the bounding sphere for some object like this:

```
longest_radius := 0.0

foreach vertex in the object
   vertex_distance := distance from vertex to object's origin
   if vertex_distance > longest_radius
      then longest_radius := vertex_distance
```

When we perform this computation, we will end up with a number that is somewhat close to the correct radius of the bounding sphere (though it will usually not be exact, not even to the precision of whatever numerical representation we are using, because error will accumulate through the compound mathematical steps we perform to find 'vertex_distance'.

Let's say, for the sake of argument, that the bounding radius we compute ends up being a little bit smaller than the right answer. You can see how, if the bounding sphere is too small, a bounding sphere filter test could decide that there is no collision, when in fact there should have been (because the little bits of the object that are poking through the sphere have collided).

It's much worse than that, though. To detect a collision between two objects, we want them to be in the same coordinate system, which means we have to push at least one of them through a transformation matrix. Unless we are using very, very precise and picky math for representing the matrix (game programmers generally won't, for reasons of speed and schedule) then by the time a vertex has been transformed, all hell has broken loose in terms of numerical accuracy (especially since the matrix isn't very accurate to begin with -- think about all the operations you perform to compose a matrix and you'll understand.) If this error ends up pushing the vertex further away from the center of the object during transformation, the discrepancy between conclusions reported by the bounding sphere filter and by the hard case will grow accordingly.

Collision detectors that are supposed to work together but end up contradicting each other are very bad, especially if you're trying to maintain any kind of system invariants. For our system, we decided that it must always be true, when the server is in the steady state, that no objects interpenetrate. Yet if the bounding sphere test causes a collision to be ignored, we might find two objects interpenetrating at the beginning of an object update. This Is Bad. It still happens in our system; we deal with it by calling an emergency routine that removes an object from the world and then tries to put it back in an arbitrary place as close to its last position as possible.

They Are Not For You

“What do you want?” can be a very difficult question to answer. In our case it turned out that, back in the beginning when we asked ourselves what we wanted, our answer was insufficient. All along while writing our collision detection routines we assumed that we would be able to just plug in some equation-solvers and have some really neat looking physics, rather than the “reverse the object’s velocity” kind of bouncing we had started with.

It turned out that, in order to maintain a physical simulation system in which awful computational mistakes did not happen, we needed to be *very* accurate about where and when collisions took place – otherwise objects would get stuck, or they would bounce themselves further into the object they were supposed to be repelling, and then go flying off into outer space at ten times the speed of light. Or worse.

To find the exact time of collision, we would perform a binary search in the time domain – if the timestep started at time *t*, and we found that objects A and B were colliding at time *(t+1)*, then we would backtrack them both to time *(t+0.5)* and test them again. If they’re still colliding, we go further backward in time; otherwise we go forward again. We stop when we find the earliest time at which A and B were still not colliding, down to a resolution of *t* that we find personally satisfying.

Once we find the collision time, we compare A and B to find the *closest features* of each object to the other. Each feature that we find to be close enough to the other object, we consider to be colliding, and pass that information to the physics system.

Let's Rock

Now we'll examine the effectiveness of the various filters in a simple game situation. The numbers were acquired in the following situation: the author joins a game server and flies a **hovertank**, using it to drop a **cargo box** onto the landscape. When the box hits the landscape, it disappears and deploys a **repair pad**, appearing in its place. The repair pad pivots and slides against the ground until it comes to rest. (Once the pad comes to rest, it is marked as *sleeping*, which means that no collision detection or physics routines will act on it until it is upset by an outside force.) The hovertank then turns around and touches down on the repair pad (in a concave area, such that there are no dividing planes), then takes off again and flies away.

entity type	# of polygons before BSP	# of polygons after BSP
hovertank	97	111
cargo box	12	12
repair pad	69	76

Here is a listing of collision tests, the number of times during the sample run that they were effective (came up with a definitive answer about the status of a collision), the percent of all cases for which they were effective, and the average amount of time taken for one test (timings taken from a Pentium 166 running Linux, program compiled with gcc –ggdb –O3 –m486).

Collision Test	Cases effectively handled	Average time per case
bounding spheres	3800	tiny
bounding box safety	1911	26.13μs
plane divides entities	93	231.94μs
BSP hard case	842	450.98μs

This diagram shows Plane Divides Entities to be a fairly ineffective, yet expensive, test. This is true, though it did not appear that way to us originally. Before we added Bounding Box Safety, PDE was much more effective than shown above (though it was still just as expensive). However, once it was added, Bounding Box Safety stole most of the easily filterable collisions away from PDE. At present it looks like we will end up junking PDE. Furthermore, if [Got96] is as good as they say, Bounding Box Safety will get even faster.

Note the high percentage of the time that the system resorted to using the BSP hard case; in some sense our test run is a pessimistic measurement since during landing the hovertank becomes intertwined with the repair pad, much more messily positioned than the average player is during the majority of gameplay. Extrapolating from these performance numbers, our system would slow down uncomfortably if all 32 players in a full game decided to land on complex structures at once. However, we believe that the system meets current needs.

The code for the collision detection algorithms themselves is not optimized; we spent all our time dealing with the software engineering intricacies of putting together a modern game. It seems likely that simple optimizations could speed up each test by several times, though we feel it would be more worthwhile to modify the algorithms.

Conclusions, Current Work, and Future Concepts

The use of easy filters and BSP trees has sped up our system drastically beyond the simple brute force approach, but there is plenty room for further drastic improvement.

BSP trees require a fair amount of computation to construct, and once built, they cannot easily be modified except in very special cases. Because of this, BSP trees are not very useful for objects that change shape. (They *can* be used for precomputed 3D mesh animations; you just precompute one BSP tree per frame of animation. That whole technique, however, has unappealing limitations.)

Several researchers have gone about creating high-performance collision detection systems. Here we summarize a few such systems and provide our own comments.

Philip M. Hubbard describes an algorithm in [Hub96] that uses a hierarchy of spheres to represent an object. This is appealing at first because sphere-sphere intersection is very cheap. Hubbard engages in expensive precomputation to find a good fit of spheres for a given object, which makes the algorithm less attractive for simulating non-rigid bodies. Also, many spheres are required to represent complex objects with reasonable accuracy, and the memory required to store said spheres becomes large. Finally, it does not seem that this algorithm will perform well for objects that are tightly intertwined – though the results presented in Hubbard's paper are hard to interpret, as they are unclear at best.

[Pon] and [Lin] describe methods for detecting collisions by tracking the closest sets of features between potentially colliding objects. Such methods are appealing because they might scale up very well: if we can isolate the features of an object which are candidates for collision, and we can navigate the objects as they rotate to incrementally consider new features, then we do not even need to transform most of an object's geometry to test it for collision in common cases. These methods generally involve building Voronoi diagrams for the shapes in question – an expensive procedure which again limits us to rigid bodies, and which is further troublesome because numerical accuracy is a notorious problem in the construction of a 3D Voronoi diagram.

More recently, [Got] et al propose using hierarchies of Oriented Bounding Boxes (OBBs) and testing for intersection between these boxes using a quick test based on a separating axis theorem (which is another way of saying that there always exists a separating plane between two convex polyhedra). The basic ideas bear similarity to Hubbard's, though the methods for approximating shapes and testing intersections are completely different. OBB-trees will generally fit a shape better than Hubbard's spheres for a given number of subdivisions, but they require much more memory, so a sphere-approximation system could use deeper trees within the same budget. The performance figures that Gottschalk et al present are impressive, though the examples they present in the paper include non-rigid objects and they neglect to show performance numbers for recomputation of the OBB-trees.

Where do we go now?

Speaking for ourselves, it seems likely that in the next major revision of our engine we will abandon BSP trees and instead use a hierarchy of bounding volumes. Rather than require the bounding volumes to fit shapes as closely as Gottschalk or Hubbard, we will probably use looser-fitting volumes with correspondingly shallower trees. Each bounding volume would contain a list of the features it encloses; once we had determined that some set of leaf-node volumes between two objects intersect, we would perform n^2 collision detection between pairs of offending volumes (though at this point n would be very small.) Both spheres and axis-aligned bounding boxes are appealing candidates for such volumes, since they do not necessitate the successive matrix operations required by OBBs.

Hierarchies of bounding volumes could be made to work quite well for dynamic shapes; if a moving feature is allowed to stretch the bounding volume that contains it, and if that stretching propagates itself up to the root of the hierarchy, then a moving shape will always have a valid bounding hierarchy which is quickly computable. (Continued validity would be assured so long as we never allow a bounding volume to shrink past its original dimensions.) The bounding hierarchy for a moving object might slowly degrade over time, but subtrees of the bounding hierarchy could be incrementally recomputed to maintain a tight fit at little computational cost.

Hierarchies which store shape features are also a natural fit to our needs, since the data structure would be equally useful for determining collision and finding closest feature sets, and in fact the procedute which tests two objects for collision might return sets of closest features as an incidental side-effect. This would be very nice compared to our current closest-feature routine, which is very slow.

References

[Chin95] Norman Chin, "A Walk Through BSP Trees", *Graphics Gems V,* AP Professional, 1995.

[FV90] J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics Principles and Practice, 2nd Ed,* Addison-Wesley Systems Programming Series, 1990. See especially section 12.6.4.

[Got96] S. Gottschalk, M. Lin and D. Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection", *Siggraph '96.*

[Hub96] Philip M. Hubbard, "Approximating Polyhedra with Spheres for Time-Critical Collision Detection", *ACM Transactions on Graphics,* Vol. 15, No.3, July 1996. Available at http://siesta.cs.wustl.edu/~pmh/research.html.

[Lin] Ming C. Lin and Dinesh Manocha, “Efficient Contact Determination Between Geometric Models”. Available at http://www.cs.unc.edu/~manocha/collision.html.
[O94] J. O’Rourke, “Computational Geometry in C”, Cambridge University Press, 1994. ISBN 0-521-44592-2 paperback, ISBN 0-521-44034-3 hardback.
[Pon] Madhav K. Ponamgi, Dinesh Manocha, and Ming C. Lin, “Incremental algorithms for collision detection between solid models”, Available at http://www.cs.unc.edu/~manocha/collision.html.
[Rab94] Rich Rabbitz, “Fast Collision Detection of Moving Convex Polyhedra”, *Graphics Gems IV*, AP Professional, 1994.
[Wade97] Bretton Wade, “BSP Tree Frequently Asked Questions”. Available at http://rtfm.mit.edu/pub/usenet/news.answers/graphics/bsptree-faq (non-authoritative copy).