Dynamic Plane Shifting BSP Traversal

Stan Melax

BioWare

Abstract

Interactive 3D applications require fast detection of objects colliding with the environment. One popular method for fast collision detection is to offset the geometry of the environment according to the dimensions of the object, and then represent the object as a point (and the object's movement as a line segment). Previously, this geometry offset has been done in a preprocessing step and therefore requires knowledge of the object's dimensions before runtime. Furthermore, an extra copy of the environment's geometry is required for each shape used in the application. This paper presents a variation of the BSP tree collision algorithm that shifts the planes in order to offset the geometry of the environment at runtime. To prevent unwanted cases where offset geometry protrudes too much, extra plane equations, which bevel solid cells of space during expansion, are added by simply inserting extra nodes at the bottom of the tree. A simple line segment check can be used for collision detection of a moving object of any size against the environment. Only one BSP tree is needed by the application. Successful usage within commercial entertainment software is also discussed.

Key words: collision detection, BSP tree, video game.

1 Introduction

Simulating an object as a point is a popular technique to reduce the complexity of various math and physics problems. Fast collision detection is important for interactive 3D applications that wish to maintain a high frame rate. Not surprisingly, one popular method of doing collision detection of an object with an arbitrary polygonal environment is to approximate the object as a point. The reason the object does not intersect the environment's geometry is because the object does its collision detection with an approximate offset surface - an "expanded" or "scaled" copy of the geometry where the interior walls have been moved inward, exterior walls shifted outward, the floors raised, and the ceiling lowered. Note that by environment we are referring to a large, detailed, 3D model that is rigid (static).

As the object moves from one position, v_0, to another, v_1, the motion line segment, (v_0,v_1), is checked against the offset surface to determine if it has collided. If a collision has occurred, there are a number of options for correcting the object's position. One possibility is to place the object at the point where the segment impacts the geometry. If the object is a freely moving physical body then its velocity can be mirrored to simulate an elastic collision. If the object is the user's avatar, the object can be easily made to slide along the plane of impact. This prevents the user from getting stuck when navigating near walls.

Note that just treating an object as a point is not a sufficient method for fast collision detection. An arbitrary polygonal environment can contain thousands of polygons. Therefore the geometry should be represented in an efficient spatial structure such as a binary space partitioning (BSP) tree.

A disadvantage of this offset surface technique is that it requires an additional copy of the environment's geometry for each object shape/size. If an object is allowed to change orientation, then there are further symmetry restrictions on the object's collision envelope. Typically, the environment's geometry is offset using a standard-sized sphere or cylinder for reference.

Our primary interest in this problem started with characters navigating a 3D environment in a video game. We do not want anything coming within a cylinder around the character. Prior to using the technique presented in this paper, the environment's geometry was offset to reduce these cylinders down to a point. In an effort to provide a content-rich game, we have many different sized characters. The memory requirement for having multiple copies of the environment's geometry was a problem. In addition to characters, our game also creates many small artifacts to make special effects such as explosions and debris. These small artifacts require fast collision detection as well. Creating another BSP tree for every particle size is just not feasible.

This paper presents dynamic plane shifting BSP traversal, a technique to overcome this single size limitation. Collision detection is still done using a fast line segment check. The environment is represented with only one standard BSP tree that was constructed without any regard for what shapes it would be doing collision detection with. We modify the plane equations of a BSP tree during the collision detection traversal. When BSP trees are constructed, we insert additional beveling nodes to limit the influence of solid volume cells when expanded. Our method will give a reasonable approximation for collision detection of an arbitrary convex shaped object moving along a linear path.

2 Related Work

Among the fastest collision detection algorithms are those specifically designed for pairs of moving convex polyhedra [Cohen95, Mirtich97]. Our environment has complex geometry. If it were broken down into convex polyhedra, there would be a large number of pieces to deal with. While our object is moving, the environment we deal with is static. Therefore, a BSP-based approach to collision detection is the best option available for our problem.

Merging BSP trees [Naylor98], or OBB trees [Gottschalk96] does provide accurate collision detection for arbitrarily shaped objects. However, the cost of such algorithms is a concern. Experimentation [Gottschalk96] has demonstrated fast collision detection (4ms) between two objects with 143690 polygons each. Unfortunately, such performance is not guaranteed. The very same research includes a smaller example with a 4780 object next to a 44921 polygon object and yet it runs 20 times slower (100ms) on a computer twice as fast. Similar work [Klosowski96] reports a series of individual query costs instead of just the averages. This reveals significant variation in the cost from one collision query to the next. The worst case happens when the objects are closer and there are more contacts. In a real application this worst case would be compounded further by the collision resolution mechanism. The collision query would have to be repeated until non-intersecting positions are found for the two objects. Video games have to maintain an interactive frame rate without any stalling. Extra time is not available when collisions/contacts occur. The majority of the CPU resources are being used by other parts of the application. Typically there are many moving objects being processed per frame. Some of these objects are in continual contact with the environment. Environments consist of tens of thousands of polygons. While our algorithm does not utilize all the details of the object's geometry, it is able to consistently meet our performance objectives. Our method is much faster since only a line segment is merged with the BSP tree.

Without additional volume extrusion over the path of travel, merging trees only produces interpenetration information. More processing is required to determine time and place of impact. With a large enough time step it is possible to pass through objects. Our technique extrudes the object's geometry (a point) from one point in time to another. This produces a line segment. A single query of this segment against the BSP tree indicates a time and place of the collision. There is no possibility of passing through matter.

The idea of using an offset surface for collision detection with an object's reference point (such as its center) goes back to early video games on the Apple 2 and Atari 2600. The combination of this with BSP trees was used in the popular video games Doom and Quake [Carmack]. Instead of having multiple BSP trees and restricting the possible dimensions of collidable objects, we accommodate different sized objects with only one BSP tree.

Accurate offset surfaces can be generated by Minkowski summation or polyhedral convolution [Basch96]. These procedures can significantly increase the number of faces. In practice, it is common to create approximate offset surfaces by moving existing faces and vertices. We were doing this previously. The error in this method is a bit worse than approximating an object with its convex hull. Normally, there are 3 non-degenerate cases of impact [Baraff97] for a pair of convex objects a and b: 1) vertex of a with face from b, 2) vertex of b with face from a, and 3) edge from a with edge from b. By treating our object as a point, the only contact that it can experience is a vertex on the object with a face of the environment. Similar errors are also present in the new technique presented in this paper. Beveling is used to improve the approximation.

Beveling geometry to limit its protrusion when scaled or expanded is not a new idea. Early polyline drawing algorithms would do this when drawing thick lines. This technique has been applied to boundary representations (the set of polygons describing a model) when expanding the geometry. From what we understand of the source code available, Quake's BSP precompiler [Carmack] does something like this to generate its offset surface for player collision. Quake's compiler then builds an additional BSP tree for this expanded geometry. Instead of the order: 1) bevel, 2) expand, 3) compute BSP tree, our order of operations is: 1) compute the BSP tree, 2) bevel, 3) expand. Because beveling comes after computing the tree, our beveling step is applied to the solid cell leaf nodes of the tree. Since expansion is the last step in this process, we are able to do this dynamically at runtime.

3 Dynamic Plane Shifting BSP Algorithm

The standard algorithm for colliding a ray with a BSP tree is a recursive function that starts at the root. If the segment lies on one side of the node's plane then the segment is passed down to the corresponding subtree. Otherwise the segment crosses the plane so it is split. The first piece of the segment is checked. If it fails to collide then the second piece of the segment is checked against the other subtree. If a solid leaf node is reached then the algorithm returns a collision with impact equal to the start of the subsegment that reached the leaf.

Here, in more detail, is our revised algorithm that dynamically alters the plane equations:

```
HitCheckBSP(node n,vector v0,vector v1)
 int hit = 0
 vector w0,w1
 if n is an empty leaf
   return 0
 if n is a solid leaf
   Impact = v0
   return 1
 if dot_product(n->normal,v1-v0) > 0
   if rayunder(n shift up,v0,v1,&w0,&w1)
     hit = HitCheckBSP(n->under,w0,w1)
     if hit==1
       v1 = Impact
   if rayover(n shift down,v0,v1,&w0,&w1)
     hit |= HitCheckBSP(n->over,w0,w1)
   return hit
 else
   same thing, but in the other direction
End
```

The function `rayunder` returns true if part of (v_0,v_1) lies under the supplied plane. The portion of the line segment under the plane (cropped if necessary) is returned in (w_0,w_1). The function `rayover` has similar functionality. Unlike the previous algorithm, the segment is not divided into two disjoint pieces - the subsegments passed down into the subtrees will overlap. Even if a collision occurs in the first subtree, it may still be necessary to check the other subtree (after adjusting the segment endpoint) since an impact may occur sooner.

Notice that the plane is translated twice when we visit a node. The offsets are specified at runtime and may depend on the plane's normal. Furthermore, the upward and downward offsets do not have to be equal. There are various options available. We describe spherical, cylindrical, and general convex expansion. If the planes are translated by a constant factor d, then the colliding object is a virtual sphere of radius d. This is implemented by simply adding (or subtracting) d to the constant component, D, of the node's plane equation $Ax+By+Cz+D=0$. It is also easy to construct a formula to fit a cylinder. The plane is offset by the dot product of the plane normal with the vector from the cylinder's reference point to where the plane rests tangent to the cylinder's rim. Spherical or cylindrical expansion is fast and adds very little overhead to the collision detection.

Another appropriate amount to offset the plane is:

$$d = \max_{v \in vertices} \{n.normal \bullet v\}$$

The maximum is taken over the vertices of the object. Obviously, the vertex that determines this maximum will be on the convex hull. Therefore it can be found in *O(sqrt(num_hull_vertices))* using gradient descent, or in *O(lg(num_hull_vertices))* by exploiting a DK hierarchy [Dobkin85]. Finding the maximum with the normal negated determines the amount to offset the node's plane in the downward direction. By using a vertex on the model, we have a point of impact (other than the object's reference point) in the event of a collision. Impulses and rendering effects can be applied to this point.

Note that we have not yet pursued rotation with our collision detection technique. With rotation the offsets would be different for point v_0 than for v_1 since they occur at different instance in time.

For collision purposes, BSP trees made from solid geometry are more efficient since the polygons can be ignored. BSP trees representing polygon soup must store the polygons at nodes in the tree. Our plane shifting gives no indication how to alter polygons, and our beveling step relies on the solid/empty status of the leaf cells. Therefore, we do not claim that our method will even work with polygon soup.

4 Why This Works

To analyze the algorithm, it helps to understand what a BSP tree represents. A BSP tree decomposes space into convex cells, which are either solid or empty. Each cell corresponds to a leaf node of the BSP tree. Its solid/empty status depends on which side of its parent's plane equation it resides. The cell is defined by the plane equations of the nodes on the path from its parent node up to the root of the tree. Some of these planes may not actually touch the cell.

In standard BSP algorithms, as the line segment (our object extruded over time) is passed down the tree to a leaf node, it is clipped according to the node's plane equations along the way. What reaches a leaf node is the intersection of the cell with the original line segment given to the root of the BSP tree.

When we move the planes, what reaches a cell will be the portion of the segment that intersects the expanded cell (planes moved outward). According to our algorithm, if anything reaches a solid cell, then a collision has occurred. The boundary of the union of all the expanded solid cells effectively simulates an approximate offset surface.

5 Potential Inaccuracy and Beveling

As mentioned previously, our technique (and similar techniques that simulate the object as a point) are subject to a certain degree of inaccuracy. This is illustrated in Figure 1. Two spheres, A and B, collide with the geometry below them. Using the center of each sphere is its reference point, the geometry is offset outward as represented by the dotted lines. The center of each sphere rebounds off these offset planes. As sphere A moves along its trajectory, the collision occurs before its perimeter contacts the surface. For comparison, sphere B will rebound with an appropriate impact point and direction of deflection.

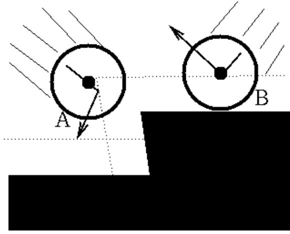

Figure 1: Collision using offsets

Most of the time, the protrusion of solid cells is tolerable, but the error starts to increase dramatically as the angle between adjacent planes approaches 180 degrees. We reduce this problem by beveling the solid cells of the BSP tree.

Figure 2 shows an example by looking at a small part of a BSP tree. Plane *C* divides a solid cell from an empty cell. The solid cell is beveled with two planes (*D* and *E*) by inserting extra nodes at the bottom of the tree.

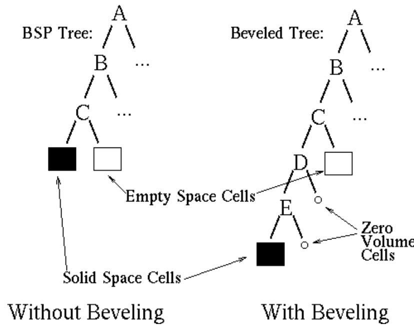

Figure 2: Bevel effect on BSP tree

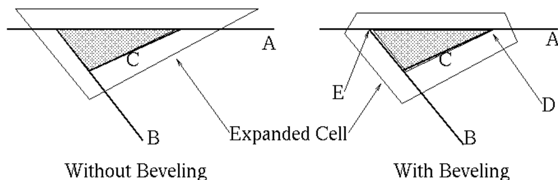

Figure 3: Bevel effect on plane shifting

Figure 3 shows the difference made by the added nodes. When the cell is expanded, it no longer protrudes as far as it did without the extra nodes.

When compiling a boundary representation into a BSP tree we are already keeping track of the volume of space at each node. This was originally done to help evaluate candidates when selecting the next partitioning plane [Berg93,Naylor98,Paterson90]. At the solid leaf nodes, these volume cells are inspected for neighboring faces (planes) that intersect at angles greater than 90 degrees. For every such case where the dot product of the two plane normals is less than zero, an additional node is inserted into the BSP tree. This added node's plane normal is half way between the two plane normals and is coplanar with the intersection of the two planes (the edge between the two faces). The resulting tree represents the same initial boundary representation, but it contains extra empty cells of zero volume. In addition to sharp edges, sharp spikes (vertices) may also need to be beveled.

Although it may not always seem necessary, this beveling step is very important, even if the initial boundary representation contains no acutely convex angles. Our initial experiments in altering the plane equations, where we had not included the beveling step, were unsuccessful. The movement of characters in our video game was blocked in the strangest places as if someone had randomly placed invisible walls in the scene. This happens because the BSP compilation process splits polygons and creates cells with sharp angles in unexpected places. Beveling solved this problem and made the dynamic plane shifting technique viable.

Additional beveling can improve accuracy, but there is the cost of adding nodes to the tree. Extra memory and performance costs are analyzed in a later section.

6 Performance Justification

We have added some additional steps to the BSP traversal. Clearly, it will be faster to have a BSP tree based on offset geometry instead of dynamically shifting planes in the BSP. The reason for using our technique instead is because the BSP tree adapts to different sizes so only one tree is needed. To justify using our technique, it remains to be shown that it maintains a performance advantage over the alternative (single tree) method, which is to merge the object's geometry with the environment's BSP tree.

We could compare performance with a complicated model for the object, but that would give our technique an unfair advantage. Our algorithm's approximation to collision detection cannot be better than checking the convex hull of the object against the environment. Therefore let us assume our object is a convex polyhedron.

So far we have been talking about collision detection of moving objects. This is easy for a point since its motion can be represented as a line segment. Extruding a 3D model would make this discussion difficult. In this analysis we give up our temporal advantage and consider a stationary object. The query asked of both algorithms is whether or not an object interpenetrates the environment.

The algorithm for merging a BSP tree with a convex polyhedron (for the purposes of detecting interpenetration) is as follows:

```
M(node n, polytope p)
  if n is leaf
    return (n is solid)?1:0
  if p is completely over n->plane
    return M(n->over,p)
  if p is completely under n->plane
    return M(n->under,p)
  else
    (pa,pb) = Slice(p with n->plane)
    return M(n->over,pa) | M(n->under,pb)
End
```

The function `Slice` partitions the polyhedron into two disjoint pieces on opposite sides of the specified plane.

To determine if a stationary object interpenetrates the environment, our algorithm checks a point (instead of a line segment) against the BSP to see if it falls into any expanded cells. (Note that now $v_0==v_1$.) Previously we have discussed our algorithm in terms of altering the planes. For purposes of comparison, it is easier to study the *dual* of our technique. It is not hard to see that our algorithm is equivalent to:

```
F(node n, polytope p)
  if n is leaf
    return (n is solid)?1:0
  if p is completely over n->plane
    return F(n->over,p)
  if p is completely under n->plane
    return F(n->under,p)
  else
    return F(n->over,p) | F(n->under,p)
End
```

The only difference between our technique, `F()`, and merge, `M()`, is that `Slice()` is not used. If any of the polyhedron lies on one side of the plane, then the entire polyhedron is passed down to the corresponding subtree. Slicing a polyhedron is significantly more expensive than the simple dot product and comparison used to determine where a vertex sits relative to a plane. Therefore, much better performance is possible by avoiding this operation.

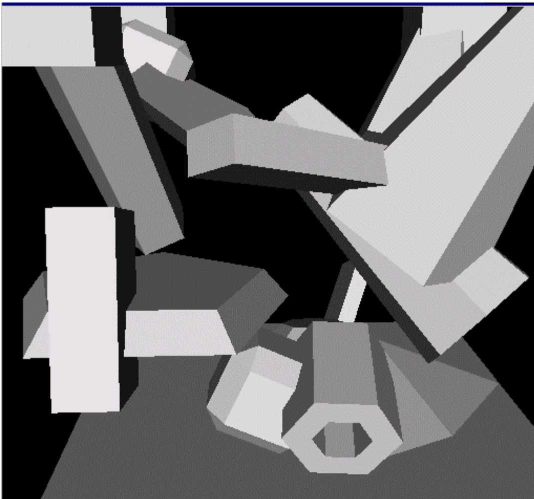

Figure 4: Sample environment geometry

We did a performance test to support our claim that slicing adds a significant cost to BSP traversal. Sample geometry for testing was made by booleaning a number of simple shapes together. The resulting 774-face boundary representation was used to generate a BSP tree with 665 nodes (including leaves). A large cube with side length equal to half the sample geometry's width was placed at the center. We tested traversal with and without the slicing step. For both algorithms we did not stop at the first intersection and return true. All intersections were computed. Times in microseconds (not milliseconds) are reported in Table 1.

Algorithm	`M()` merge	`F()` no slice
Time	14312	261

Table 1: Cost of slicing vs. not slicing

The no-slice method, `F()`, finds all cells that intersect the cube. There is the possibility of error in that additional cells may be reported. Merging the cube with the environment computes exact interpenetrating information. Its equivalent to doing a solid geometry boolean operation. Merge performed 109 slices. We slice convex polyhedrons by creating a duplicate and then cropping them both. To compensate for this duplicate effort, the time reported for the merge technique is half of its actual time. We tried to make our crop algorithm efficient. Compiler optimization was enabled. Better performance may have been possible with more low-level optimization. The 50-fold difference in time seems to indicate that slicing has a significant cost.

The entire analysis in this section has been possible since we only considered stationary objects. Another strong argument for dynamic plane shifting is that it is trivial to generalize to moving objects - the point becomes a ray. Working with a moving volume requires either adding timesteps to maintain accuracy and resolve collisions, or adding a system of extruding a volume over the motion path that can extract a time of impact when a collision occurs. Both options would require significant overhead. Clearly, better performance is possible by approximating the object as a point.

7 Performance Overhead

Using the sample geometry from the previous section, we perform some more tests to measure the beveling and tree traversal overheads. As shown in Figure 5, the number of nodes added because of beveling increases as we decrease the allowable angle between cell faces. At a 90-degree limit we double the number of nodes. The impact on performance is minor since the collision detection algorithm is in *O(lg(n))*. Performance of standard ray and point intersection tests (for zero volume objects) will not increase by more than a few percent.

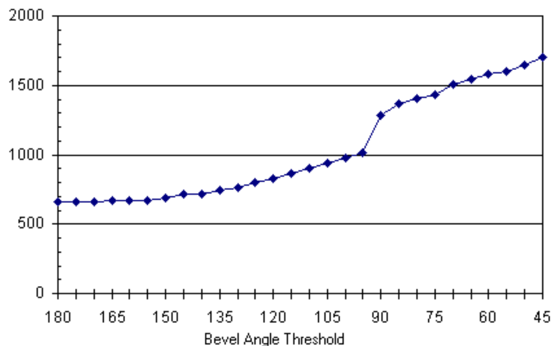

Figure 5: Size of tree vs. bevel angle

For objects, which use the dynamic plane shifting technique, there is an added cost. Additional nodes will be visited due to the bi-directional shifting of the node's plane equations. Using our sample environment, the number of nodes visited for a given object size is shown in Figure 6. For each given diameter, a number of stationary spheres were generated randomly within the volume. The average number of nodes visited during BSP collision check is reported. The diameter is expressed as a percentage of the size of the environment. As expected, the number of nodes visited increases with the given size. An object as large as the environment will touch every node in the tree. Clearly dynamic plane shifting works best for colliding objects that are small relative to the size of the environment.

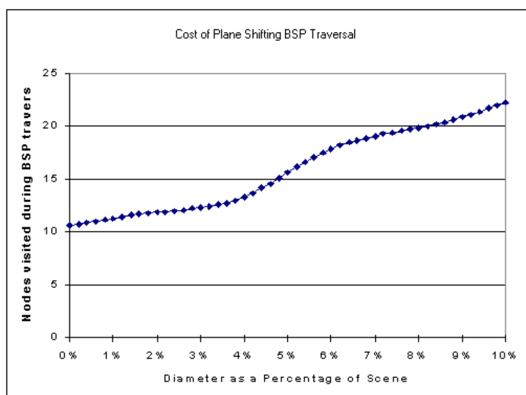

Figure 6: Nodes visited vs. object size

8 Application

We have employed our dynamic plane shifting algorithm in MDK2 [Bioware2000], a 3D video game our company will soon be publishing. It replaced our previous system that used multiple BSP trees. The movement and environment interaction of the characters is the same as before. Furthermore, we are now able to introduce more characters and objects of varying sizes, Figure 7.

Figure 7: MDK2 characters

Solid cells were beveled to ensure at most a 95-degree difference in adjacent planes. With this beveling threshold, an extent can still be expanded further than the amount the planes are moved. Since our technique is an approximation, it is beneficial to minimize error where it is most important. There is gravity in MDK2 which means the characters move along the ground. Therefore, for any solid cell that was not topped with a nearly level plane, we added a node with plane normal `z=1` and made it concurrent with the highest vertex in that cell. (Note, Quake's system also adds axial planes to its brushes.) The extra horizontal beveling plane eliminated very rare but obscure cases where a solid cell would protrude through the floor when expanded. Gameplay improved as well. A character can now walk along a ridge without constantly fighting gravity even when the ridge was constructed without a flat top.

Even with all the beveling there still is some error in the system. Measuring a difference between the true offset surface and the apparent collision surface from plane shifting would not give a practical evaluation of our technique. The game was subjected to extensive review by internal testing, by our publisher's beta testers, and by Sega's quality assurance department. There were some cases where artwork had been modified to improve accuracy. Overall, in MDK2, our dynamic plane shifting technique achieved the quality standards required for a published entertainment title.

Our trees were typically between 2 and 2.5 times the size of the non-beveled tree. This is comparable to the results from the small sample geometry that was presented in the previous section. Having a larger tree is worth the extra space cost. With over a dozen different sized characters in the game, storing an extra tree for each character would not have been feasible.

When we replaced our multiple tree collision detection with our dynamic plane shifting solution, we did not detect a difference in the overall performance of the program. Most of the computing resources go into other areas such as animation, AI, physics, and rendering. Collision detection is a small fraction of the work. Furthermore, many of the BSP queries are for particle, bullet, line-of-sight, lens flare, and shadow checks. These require little or no geometry expansion. The variance in the overall time per frame made it impossible to measure a performance overhead this way.

It was necessary to measure the performance times in isolation to provide a clearer picture of the overhead of the dynamic plane shifting technique. We present results using the player's characters, which are larger than most objects and consequently should have higher overheads. Movement updates for the characters occur each frame. In the part of the code where the player's collision check is made, we inserted code for 3 methods of collision detection: regular BSP collision (labeled Ray), spherical offset, and cylindrical offset. Each method received the same input parameters, including the player's current position and desired new position for that frame. To ensure accurate timing, each method repeated its calculation 100 times every frame. Nothing but BSP traversal code contributed to the times. The game was played for at least 2 minutes with the player character constantly moving. The average times for each method were recorded. This was done for 3 finished levels within our game. The reason for using completed artwork is because early prototype content tends to contain less detail that would have resulted in fewer BSP cells being reached by an expanded volume. Testing was done on a Pentium 3 PC computer. The algorithms tested were all unoptimized C code. The average times in microseconds of each technique for the 3 tests is shown in Table 2.

Character	Ray	Sphere	Cylinder
Max the Robot	20	51	66
Doctor Hawkins	11	27	38
Kurt Hectic	19	44	64

Table 2: Average collision query time (microseconds)

As expected, the previous method is faster than dynamically shifting the plane equations. The bounding cylinders had less volume than the bounding spheres and yet they did not perform as well. This is probably due to the cylinder offsets being computed on the fly. Our results here show that the collision detection can be 2.5 to 3.5 times more expensive. Putting these results in context, at 30 FPS, each frame allows 33000 microseconds. We felt an additional 20 to 50 microseconds for a character's collision detection was worth the flexibility of allowing different sized characters and objects in our game.

Since this is an interactive application, the collision detection algorithm must provide consistent performance. Good average times are meaningless if the worst case is orders of magnitude more expensive. Figure 8 shows the collision times over a series of 200 frames where the Doctor character was running and jumping around in a room with stairs, corners, ledges, and other objects. There is variation in the sample times, but it is within a small constant. These fluctuations are not large enough to affect the framerate of the program.

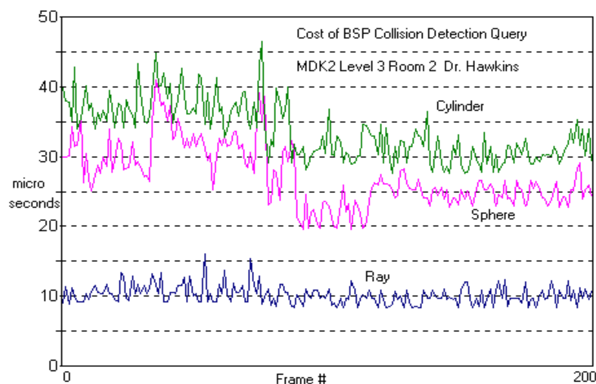

Figure 8: Variation in algorithm performance

Introducing dynamic plane shifting had other benefits. Previously we were having problems with offset surfaces. Our artists preferred to model game content in 3D Studio Max. Unlike Quake editors, which build content with intersecting convex brushes, we generate our BSP trees from 2-manifold geometry. Offsetting a boundary representation often creates self-intersections that corrupt our BSP compiler. Using dynamic plane shifting replaced this system and proved to be more robust.

As CPU speeds improve, it will eventually be practical to use a more accurate algorithm, such as merging BSP trees, for important collision detection tasks such as characters colliding with the environment. Using the object-as-a-point approximation will still be useful for less important artifacts in a 3D application. For example, our system can also support collision detection for hundreds of non-zero volume particles while maintaining the desired frame rate.

9 Conclusion

This paper addresses the problem of providing fast detection of various sized objects colliding with a static polygonal environment. Representing the object as a point makes it possible to do collision detection much faster than having to deal with the object's geometry. We no longer require an extra copy of the environment's geometry for every possible object size. Dynamic shifting of the plane equations during BSP tree traversal lets us adapt to any sized object. Beveling solid cells prevents geometry from protruding too far. Since the object is represented as a point, there is some inaccuracy in the collision detection. This error also existed in the previous system where we used multiple BSP trees. We have shown that our technique maintains a performance advantage over merging. In our practice, we have found the technique to be as accurate as using multiple trees, without losing the ability to do fast collision detection required by our application.

10 Future Work

Our dynamic plane shifting technique only addresses the issue of linear translational movement. General 3D rotation is becoming more commonplace in 3D applications. A change in orientation poses some challenges. The amount that the planes are offset depends on the orientation. If the orientation changes during a collision query then so can the offset. Currently, our algorithm does not take this into account.

The performance overhead of our technique is mostly due to the extra nodes visited. This is influenced by the object size with respect to the size of the cells of the BSP tree. For larger objects it may not be necessary to go to the entire depth of the tree. Multiresolution with BSP trees has already been used in other applications [Naylor98]. It may be possible to apply similar simplification to our collision detection algorithm.

Doing solid geometry boolean operations on BSP trees can provide a modifiable environment experience for the user. This technique will not work when multiple BSP trees are used to represent the environment. Since plane shifting BSP traversal only uses one tree, it may be possible to allow dynamic modification provided the result of the boolean can be beveled as necessary.

Acknowledgements

Special thanks to John Carmack of Id Software for confirming our description of Quake's system. David Falkner wrote the cylinder expansion routine and helped with the performance testing. Credit also goes to the rest of the MDK2 team at Bioware for testing the practical application of this technique. This paper would not have been possible without the help of Jean Melax.

References

[Baraff97] David Baraff, Andy Witkin. *Physically Based Modeling: Principles and Practice.* SIGGRAPH Course notes, 1997.

[Basch96] J. Basch, L. Guibas, G. D. Ramkumar, L. Ramshaw. *Polyhedral Tracings and their Convolution.* Algorithms for Robotic Motion and Manipulation, 1996.

[Berg93] M. de Berg, M. de Groot, M. Overmars. *Perfect Binary Space Partitions*. Canadian Conference on Computational Geometry, 1993.

[Bioware2000] Bioware, Shiny, and Interplay: *MDK2.* Sega and PC Video Game, 2000.

[Carmack] John Carmack, M. Abrash. *Quake's BSP compiler.* Id Software.

[Cohen95] Jonathan D Cohen, Ming C Lin, Dinesh Manocha, Madhav Ponamgi. *I-Collide, An Interactive and Exact Collision Detection System for Large-Scale Environments.* ACM Symposium on Interactive 3D Graphics (I3DG), pp 189-196, 1995.

[Dobkin85] D. Dobkin, D. Kirkpatrick.: *A Linear Algorithm for Determining the Separation of Convex Polyhedra.* Journal of Algorithms 6, pp 381-392, 1985.

[Gottschalk96] S. Gottschalk, M. C. Lin, D. Manocha. *OBB-Tree: A Hierarchical Structure for Rapid Interference Detection.* SIGGRAPH, pp. 171-180, 1996.

[Klosowski96] J. Klosowski, M. Held, J.S.B. Mitchell, H. Sowizral, K. Zikan: *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs.* SIGGRAPH Visual Proceedings, 1996.

[Mirtich97] B. Mirtich: *V-Clip.* Technical Report 97-05, Mitsubishi Electric Research Laboratory, 1997.

[Naylor90] Bruce F. Naylor, John Amanatides, William Thibault. *Merging BSP Trees Yields Polyhedral Set Operations.* SIGGRAPH, pp 115-123, 1990.

[Naylor92] Bruce F. Naylor. *Interactive Solid Modeling via Partitioning Trees.* Graphics Interface, pp 11-18, 1992.

[Naylor98] Bruce F. Naylor. *A Tutorial On Binary Space Partitioning Trees.* Computer Games Developer Conference Proceedings, pp 433-457, 1998.

[Paterson90] Michael S. Paterson and F. Frances Yao, *Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling*. Discrete and Computational Geometry, vol 5, pp 485-503, 1990.

[Torres90] Enric Torres, *Optimization of the Binary Space Partitioning Algorithm (BSP) for the Visualization of Dynamic Scenes*. Eurographics 1990.