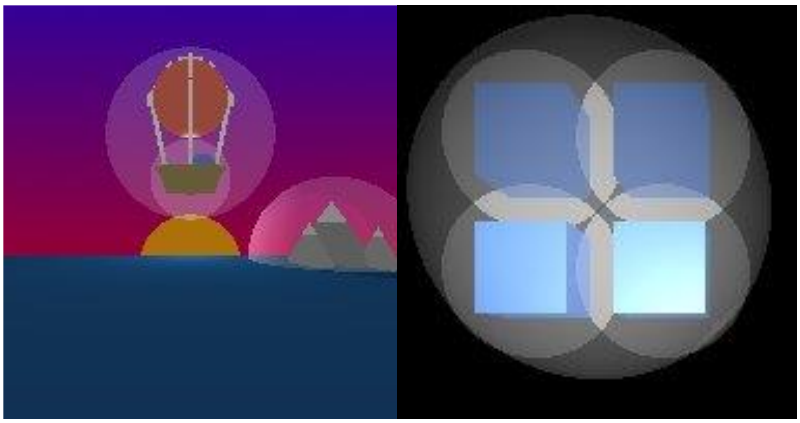
Collision and Interference Detection using Dynamic Hierarchical Bounding Volumes - Sept 2001

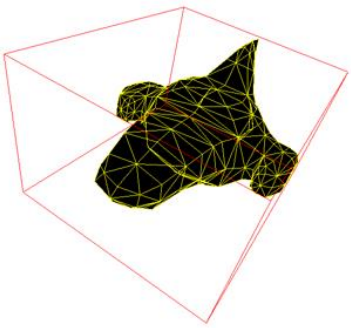
Overview:

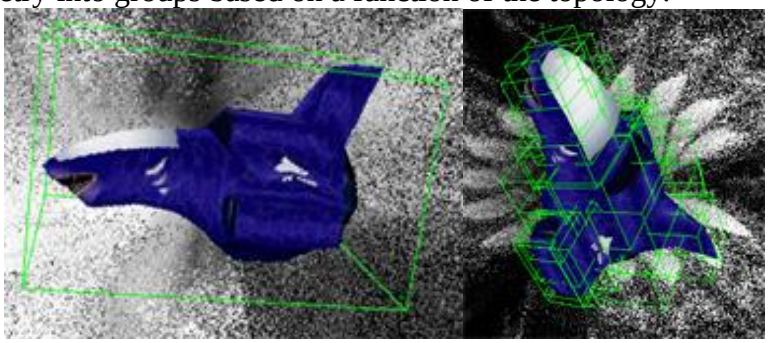
The definitive factor in making a great hardware-accelerated 3D interactive game is fast collision detection (CD). With the industry gearing towards removing the bottleneck of the 3D pipeline, CD has now become the main factor that can make or break a game. With more realism and a huge number of polygons, gamers want the most immersing and surreal experience possible. This means thousands of possible collisions hundreds of times every second that must be checked and dealt with.



<- Should Spheres be the bounding volume of choice? Lightning fast intersection testing would seem to testify such a notion, but that is not the complete story.

Objects are represented as triangles for purposes of pipelining to the graphics card. As such there is a natural inclination to do collision detection on those polygons. An optimal algorithm will not perform all pair-wise triangle intersection tests, instead organizing the triangles representing the geometry into groups based on a function of the topology.

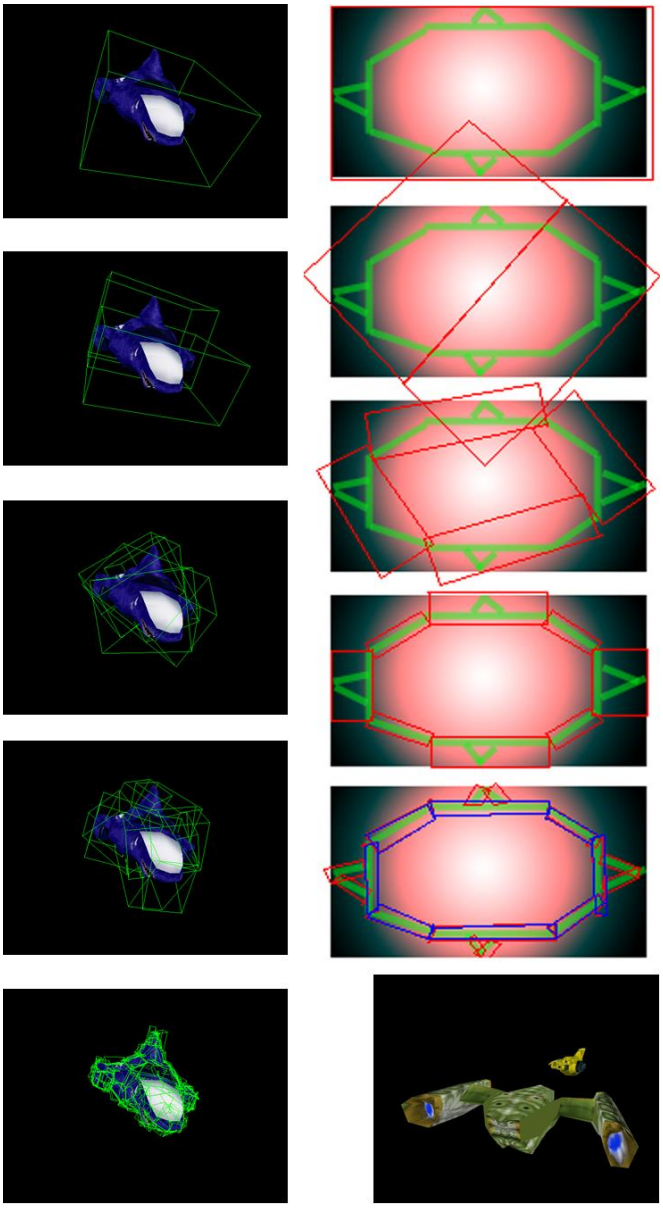




AABBs and OBBs offer accurate collision but at a higher price. Is it worth the risk? (Axis-Aligned and Oriented Bounding Boxes; AABB, OBB)

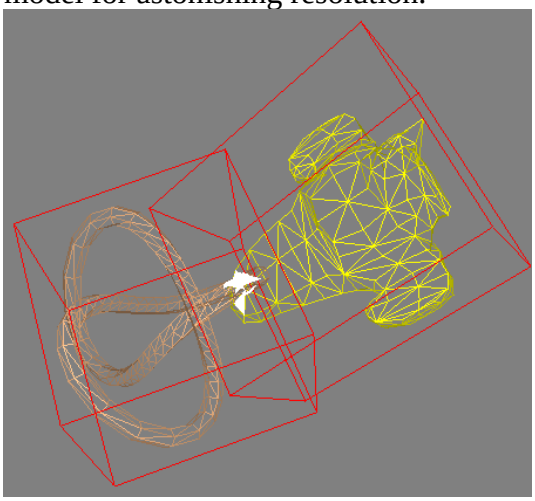
Starting from a list of possible collides we perform CD on them in an organized manner. Each CD routine call is fully customizable, allowing as much accuracy as needed by the client. Possible objects for interference detection (ID) include players, projectiles, terrain chunks, lights, the frustum, stray rays, forces, and sounds modeled as rays. Simple geometric shapes are used to represent all entities, turning the CD routine into a trivial task.

The Dynamic Hierarchical Bounding Volume Algorithm- The Basics:



The cornerstone of the collision detection for CAS is being able to represent the subs and terrain patches using bounding box (BB) trees. Pictured to the far left here is a geometric description of the hierarchy construction for a sub. Just to the right of it is a similar yet more intuitive BB creation process (hand-drawn). Notice how each iteration brings the BBs closer to the actual shape of the geometry. The trees are dynamic sphere/box trees, allowing various levels of accuracy for CD.

Not only that, but CAS boasts the most accurate CD possible-at the polygon level. At the leaves of each bounding hierarchy, the bounding volumes (BVs) contain polygons, exploiting the input model for astonishing resolution.



Collision Level of Detail allows for optimum performance. Ranging from trivial acceptance/rejection spheres, all the way down to per-polygon ID, the programmer is given full freedom.

Conventional Collision Detection methods may have a variable number of BVs at each level. However, our method uses a binary tree structure, which always has two BVs at most per level. This gives our CD routines O (lg (n)), while the other guys have O (m^2*lg (n)). Our collision routines are asymptotically better than traditional methods, with actual worst case run time performance of lg (n) (times a small constant), with n being the total # of BVs in both objects that CD is being performed on. A formal proof is within the scope of this document, but is trivial enough that it will not be derived in lieu of more important topics to cover.

The Meat and Potatoes of the BV Algorithm:

$$A^i = \frac{1}{2}|(\mathbf{p}^i - \mathbf{q}^i) \times (\mathbf{p}^i - \mathbf{r}^i)|$$

Area of a triangle w/ points p,q,r

$$A^H = \sum_i A^i$$

Total area of all the triangles of an object

$$\mathbf{c}^i = (\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i)/3$$

The geometric center of a triangle

$$\mathbf{c}^H = \frac{\sum_i A^i \mathbf{c}^i}{\sum_i A^i} = \frac{\sum_i A^i \mathbf{c}^i}{A^H}$$

The center of mass of all the triangles; the weighted geometric mean

$$\mathbf{C}_{jk} = \sum_{i=1}^{n} \frac{A^i}{12A^H}\left(9\mathbf{c}_j^i\mathbf{c}_k^i + \mathbf{p}_j^i\mathbf{p}_k^i + \mathbf{q}_j^i\mathbf{q}_k^i + \mathbf{r}_j^i\mathbf{r}_k^i\right) - \mathbf{c}_j^H\mathbf{c}_k^H$$

The covariant matrix calculates a weighted distribution of the vertices of an object, with the area of each triangle as the weight. We use the three eigenvectors of the symmetric covariant matrix as the coordinate system of the Oriented Bounding Box.

The algorithm was pioneered by academics in the Siggraph paper [OBBTree]. With a new CD routine for Oriented Bounding Boxes (OBBs) that is an order of magnitude faster than any previous, it is now a viable option to use OBBs in a real-time game environment. Unlike Axis Aligned Bounding Boxes (AABBs), which need sides parallel to the coordinate system of the world, OBBs can have any arbitrary orientation, allowing objects to have a much better fit.

Shown above is the bulk of the algorithm. Once the covariant matrix is obtained and you have the three normalized eigenvectors (EV), another matrix is created which is used as the basis for the OBB, with the columns of this matrix being the three normalized EVs. For a recap, an eigenvector U of a matrix M is a vector which when multiplied by M will return a scalar multiple of U (M*U=k*U with k the constant scalar also known as the eigenvalue). The new basis vectors of the OBB are scaled until they encapsulate all of the geometry.

In order to split the OBB to calculate its children, we take the center of mass of the triangles, and slice the OBB at that point along whichever axis of the OBB is longest. Other axes are chosen if the first axis chosen cannot subdivide the geometry. The OBB has no children if it is not divisible.

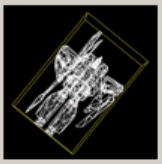
Pictured to the right is M.C. Lin, one of the creators of the BV algorithm. At the forefront of collision technology, she gave an invited lecture at the Game Developers' Conference in 2000. S. Gottschalk, now working at NVIDIA, also co founded the BV algorithm. The point is that this is a solid algorithm with years of research, proven to work in theory and in implementation (OBB trees for CAS are created offline-using [RAPID], and stored into files).

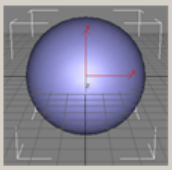
```
// Pseudo code for OBB tree creation:
OBB Build_OBB_Hierarchy( TRIANGLE_LIST Triangles )
{
   MATRIX Covariant = calculate_covariant_matrix( Triangles );
   MATRIX Basis     = calculate_eigenvector_matrix( Covariant );
   OBB BestFittingBox = refit_OBB( Basis, Triangles );
   TRIANGLE_LIST splitTLeft  = SplitOBB_Left( BestFittingBox, Triangles );
   TRIANGLE_LIST splitTRight = SplitOBB_Right( BestFittingBox, Triangles );
   BestFittingBox.LeftChild  = Build_OBB_Hierarchy( splitTLeft );
   BestFittingBox.RightChild = Build_OBB_Hierarchy( splitTRight );
   Return BestFittingBox;
}
```

The structures of the Post BV Algorithm-

Wrapping things up with the OBB/Sphere Tree:

```
typedef double fpt; // OR typedef float fpt;

// Two allocations for every obbTree: One for the vertex pointers, the other
// for the boxes.  This is possible because the number of boxes and the number
// of triangles for every object is known in advance.

struct obbTree   <- The Main Bounding Volume Hierarchy
{
   // the number of total boxes in the tree
   int boxes;

   // a chunk of memory at least as big as the number of bytes
   // required to store all the boxes for this tree
   struct box * pool_boxes; <- Memory is managed in huge chunks,
                               indistinguishable from a
   // the number of total triangles in the tree built in memory manager.
   int triangles;

   // a chunk of memory at least as big as the number of bytes
   // required to store 3 double pointers for every triangle in
   // the tree
   fpt ** pool_triangles;

   ////////////////////////////////////////////////////////////

   // the top level box for this tree
   struct box * root;              <- The BV hierarchy is implemented
                                      using a binary tree structure.
   // the center of the top-level sphere of the obb in model space
   fpt centerX;
   fpt centerY;
   fpt centerZ;

   // the radius of the top-level sphere of the obb in model space
   fpt radius; <- The top level sphere can be used as either a
                  minimum or maximum bounding sphere
};
struct box    <- The Primitive Geometric Shape (dynamic OBB/Sphere)
{
   // R0C0 R0C1 R0C2 R0C3
   // R1C0 R1C1 R1C2 R1C3
   // R2C0 R2C1 R2C2 R2C3

   // contains the rotation of the obb in the model space of the object
   // it is representing
   fpt rotationR0C0;  <- OBB          An OBB from ->
   fpt rotationR0C1;                  Jeff Lander's
   fpt rotationR0C2;                  Gamasutra Article
   fpt rotationR1C0;                  [Jeff]
   fpt rotationR1C1;
   fpt rotationR1C2;
   fpt rotationR2C0; fpt centerX; <- Sphere
   fpt rotationR2C1; fpt centerY;
   fpt rotationR2C2; fpt centerZ;

   // contains half the measure of the side length of the
   // obb for each of the three parallel sides of the box
   // in the reference frame of the box
   fpt extentsX; // = R0C3   <-OBB Extents
   fpt extentsY; // = R1C3
   fpt extentsZ; // = R2C3

   // the radius for the sphere of the obb, which is the largest distance
   // from the center of the obb to one of its diagonals, in the model space
   // of the object
   fpt radius;                        A sphere from ->
                                      Discreet's (c)
   struct box * left;                 3DS MAX 4

   struct box * right;

   struct box * parent;

   // the number of triangles in this obb,
   // which is only valid if !left && !right
   int triangle;

   // pointer to an array of vertex pointers for the triangles
   // which is only valid if !left && !right
   fpt ** vertex; <- The triangle vertex pointers for triangles at the
                     lowest level
};
```





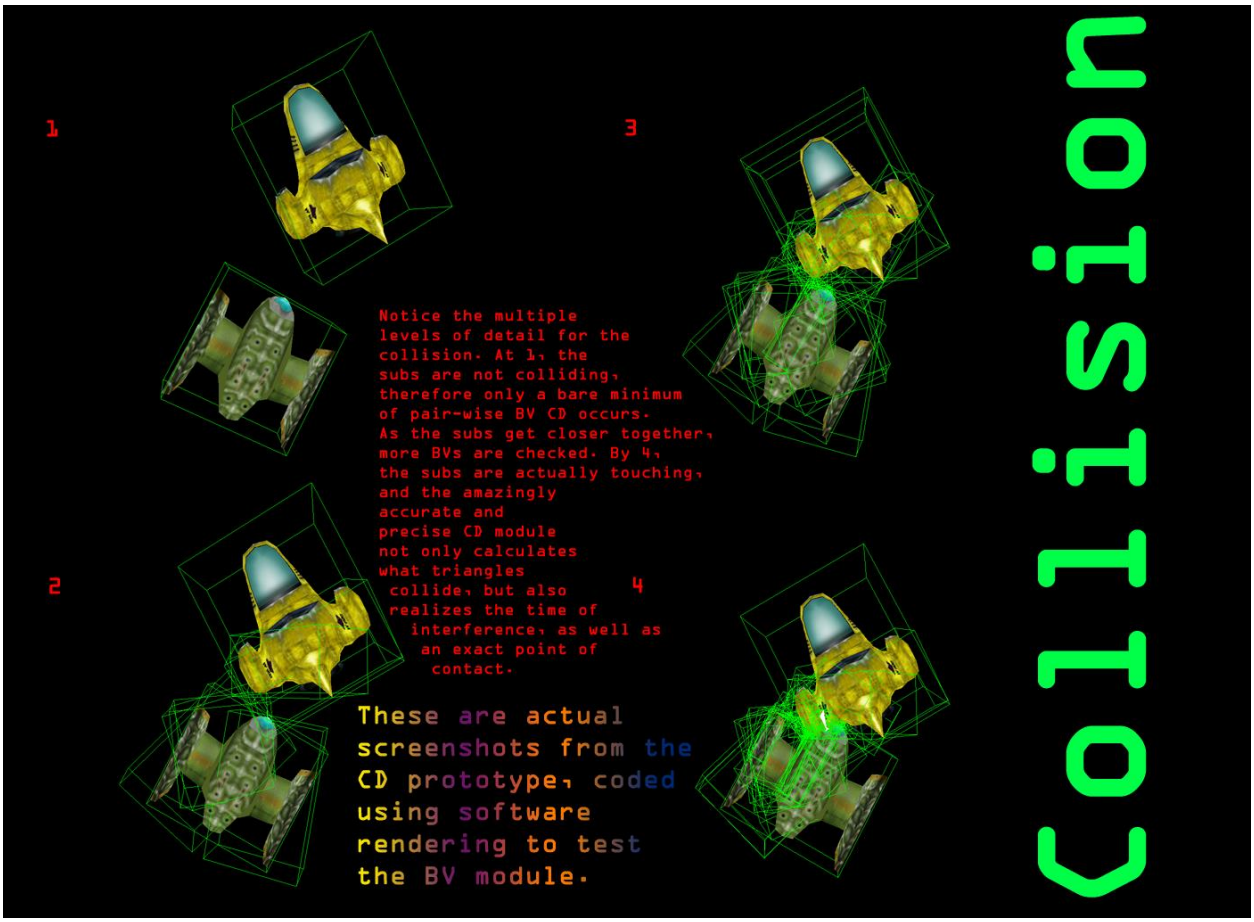
The BV structure could in theory take a variable number of different primitives. In our case, the sphere/OBB paradigm will dominate, and so the internal makeup of a BV is as such. Internal nodes contain an individual sphere/OBB, and the internal nodes together form the BSP tree hierarchy.

Notice the vertex component in the box structure. An array of vertex pointers is associated with every leaf node. The box will maintain a handle to the model's triangle vertices, saving memory and allowing a relationship to exist between the model and the BV. This may break normal modular software engineering practices, but provides the extra functionality to the collision module required to let the client know the id of colliding triangles, which is more important than just knowing their vertices.

The Last BV Section; the In-Game BV Pipeline- Collision Detection, Traversal Design, and the BV File Format:

There are three categories of CD, namely:

Bullet CD	Sub and Projectile CD	Secondary CD
Bullet-Sub	Sub-Sub	Sphere-Sphere
Bullet-Terrain	Sub-Terrain	Sphere-OBB
Bullet-Projectile	Sub-Projectile	Sphere-Frustum
Since bullets will be the default weapon, as well as being the fastest rapid-fire weapon, custom CD routines will be developed to check collision between bullets and other pieces of geometry. Moving bullets will be modeled as directed ray line segments. It will be assumed that bullet collisions may occur thousands of times a second.	Projectile-Terrain	Sphere-Ray
	Projectile-Projectile	Ray-Triangle
	The most versatile CD routines in CAS, the Sub/Projectile CD use the generality of the BV hierarchy, allowing all the above pair-wise checks to use the same code. Robustness surely follows this centralized coding scheme, allowing enough time for full debugging and in-code documentation.	Ray-OBB
		Point-Rounded_Cone
		Triangle-Triangle
		OBB-OBB
		OBB-Frustum
		OBB-Rounded_Cone
		Frustum-Rounded_Cone



```
// PseudoCode for Bullet-BV intersection
// -> for Bullet-Sub/Terrain/Projectile
BOOL Bullet_BV_IntersectionTest
(
   POINT  RayStart, // the initial point of the ray segment in world space at time t0
   POINT  RayEnd,   // the end point of the ray segment in world space at time t1
   BV     Tree,     // the hierarchical bounding volume to check intersection against
   MATRIX M0,       // the transformation matrix from world to model space for the BV at time t0
   MATRIX M1,       // the transformation matrix from world to model space for the BV at time t1
   FUNCTION ( Transform )( TIME, MATRIX ), // the function allowing us to check exact CD
                                           // -> at a possible time of intersection
   TIME  ReturnTime,     // optional pointer to store the normalized time of intersection
   POINT CollisionPoint, // optional array to store an actual point of intersection
   REAL Accuracy         // normalized value between 0 and 1 describing the intensity of the CD
                         // -> required by the client
)
{
   // Store the matrices in case the client wants to check collision again
   // -> in the same frame of reference.
   if( M0 AND/OR M1 )
      StoreMatrix( M0, M1 );

   // Scales the Accuracy by the level of depth in the BV.
   ScaleCollisionAccuracy( Accuracy, Tree );

   // Transform the bullet into the model space of the BV.
   TransformPoints( RayStart, M0 AND RayEnd, M1 );

   // Calculate the sphere of the bullet over the time intervals
   // -> to allow trivial sphere collision checks between the
   // -> bullet and the BV.
   SPHERE EncompassingSphere=CalculateSphere( RayStart, RayEnd );

   // Do the trivial rejection encompassing sphere check.
   if( TrivialRejection( EncompassingSphere, Tree ) == TRUE )
      return false;
   else
      PushStack( Tree.Children() );

   do
   {  // Perform CD with every child (depth first) until
      // collision at the lowest level has occured.
      BV_CHILD Child = StackGet();

      // Do trivial sphere-sphere CD for every level above the lowest level.
      if( TrivialAcceptance( EncompassingSphere, Tree ) == TRUE )
      {
         // If we are at the lowest level, then perform more accurate OBB CD.
         if( LevelOfChild( Child ) == Accuracy || MaxLevel( Child ) == TRUE )
         {
            MATRIX M2 = GetPossibleCollisionTimeAndTransform( EncompassingSphere,
               Tree, ReturnTime, Transform );

            // If CD at the lowest level occurs, then return the time and
            // point of collision, for physics and similar modules.
            if( ComplexCollision( RayStart, RayEnd, M2, Tree ) == TRUE )
            {
               SetTimeOfCollisionAndPointOfCollision( ReturnTime, CollisionPoint );
               return TRUE;
            }
         }
         else
            PushStack( Child.Children() );
      }
   }while( !StackIsEmpty() );

   return false;
}
```

CD is performed in the model space of the BV. Normally, checking collision between two moving objects would require solving a degree 4 equation (a + bx + cx^2 + dx^3 + ex^4 = 0), but we can approximate the movement as linear, transforming the equation into a quadratic (a + bx + cx^2 = 0). This reduces the order of magnitude of the problem by 2. Not only that, but the BV hierarchy and all of its nodes stay untouched. None of the hierarchy needs to be transformed, by the mere fact that the bullet is now in the BV's reference frame.

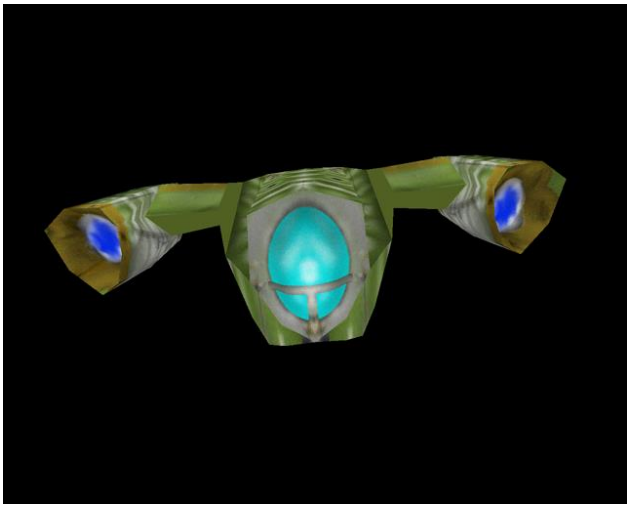
Trivial Calculations, as Desired.

Function Run Down:
-> As the function begins, we take the ray segment that the moving bullet traces out, and form a sphere completely enclosing that ray.
-> After a trivial sphere rejection, we traverse down the BV depth-first, doing pair-wise CD between the bullet sphere and the sphere of the BV nodes, until we reach the lowest level of the BV that the Accuracy variable will allow. If collision does not occur before the lowest level, then there is no collision between the bullet and the BV, and the function returns false, letting the client know not to worry.
-> If there is a collision at the lowest level, we perform a sphere sweep to get an approximate time T1 of collision, and then do two things. Taking the original two points of the bullet, we approximate the position of the bullet at the time T1 using an affine combo ((1-T1)*RayStart+T1*RayEnd). Then we use the Transform Function to get the world to model transformation of the BV at time T1. Then:
-> We apply that matrix to the new bullet point, then perform Point-OBB CD with the new bullet point and the OBB of the lowest level BV node. If the point is in the OBB, and the Accuracy is set to maximum (otherwise returning true), we perform ray-triangle with the BV-space bullet-ray and the triangles in that OBB. If the point of collision is close enough to the new bullet point, then collision has occured, and we return true with the point of collision and time.

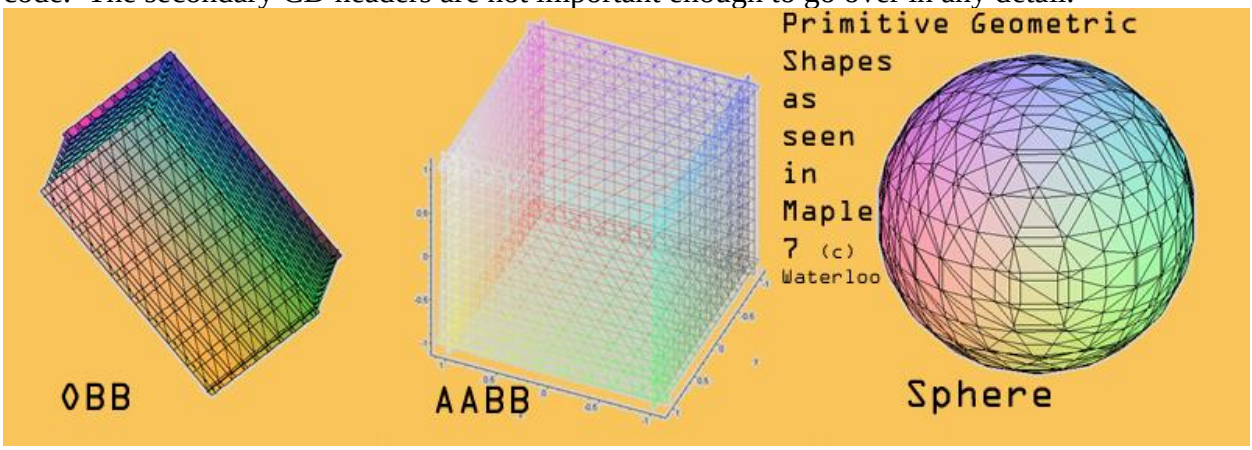
For less accurate low level collision, the Tranform function does not need to be called, and you can check Ray-Triangle without calling Point-OBB. This can be relatively faster and still very accurate.

This pseudo code outlines the basic structure of all the main CD routines, so this pseudo code should be all that is required to understand how all of the other CD functions work.

```
// Pseudo header for BV-BV intersection:
// -> for Sub-Sub/Terrain/Projectile
// -> also for Projectile-Terrain/Projectile
// Checks collision between two objects
// moving simultaneously
// during a time interval.

BOOL BV_BV_IntersectionTest
(
   BV     TreeA,
   MATRIX MatrixTreeATimeStart,
   MATRIX MatrixTreeATimeEnd,
   FUNCTION ( TransformTreeA )
       ( TIME, MATRIX ),
   BV     TreeB,
   MATRIX MatrixTreeBTimeStart,
   MATRIX MatrixTreeBTimeEnd,
   FUNCTION ( TransformTreeB )
       ( TIME, MATRIX ),
   TIME  ReturnTime,
   POINT CollisionPoint,
   REAL Accuracy
);
```



The secondary CD is used for the graphics engine and for the photo realistic preprocessed vertex color calculations. The specific things they are used for are explained in their respective TDD sections. The actual functions are trivial enough to not have to explain them in pseudo code. The secondary CD headers are not important enough to go over in any detail.



There is an intimate relationship between the sphere, AABB, and OBB. If we compare the relationship of the volume of the primitives with the speed of their respective CD routines, we find that there is a symbiotic relationship between the three.

# Of operations	Sphere	AABB	OBB
Compare	1	6	15
Add/Sub	6	12	60
Multiply	4	0	4
Abs	0	0	24

This table shows that sphere CD is nearly an un-asymptotic order of magnitude faster that AABB CD, and two orders of magnitude faster that OBB CD. Therefore, it would be optimal to rely on spheres for most of the geometry CD, and as we traverse down the BV hierarchies looking for more accurate (but fewer) collision checks, to morph from sphere CD to OBB CD. We already know that OBBs and AABBs are tighter around the representative geometry, but let

us compare the representative worst-case volumes from the reference of the OBB extents (l=length, w=width, h=height).

Assumption	Sphere Volume	AABB Volume	OBB Volume
l=l, w=w, h=h	$\frac{4\cdot\pi\cdot(w^2+h^2+l^2)^{3/2}}{3}$	$\frac{(w+h+l)^3}{27}$	$h\cdot l\cdot w$
l=x, w=x, h=x	$4\cdot\pi\cdot\sqrt{3}$	1	1
l=d*x, w=x, h=x	$\frac{4\cdot(d^2+2)^{3/2}\cdot\pi}{3\cdot d}$	$\frac{(d+2)^3}{27\cdot d}$	1
l=d*x, w=e*x, h=x	$\frac{4\cdot(d^2+e^2+1)^{3/2}\cdot\pi}{3\cdot d\cdot e}$	$\frac{(d+e+1)^3}{27\cdot d\cdot e}$	1

Equation Format courtesy of the TI (c) 92 calculator

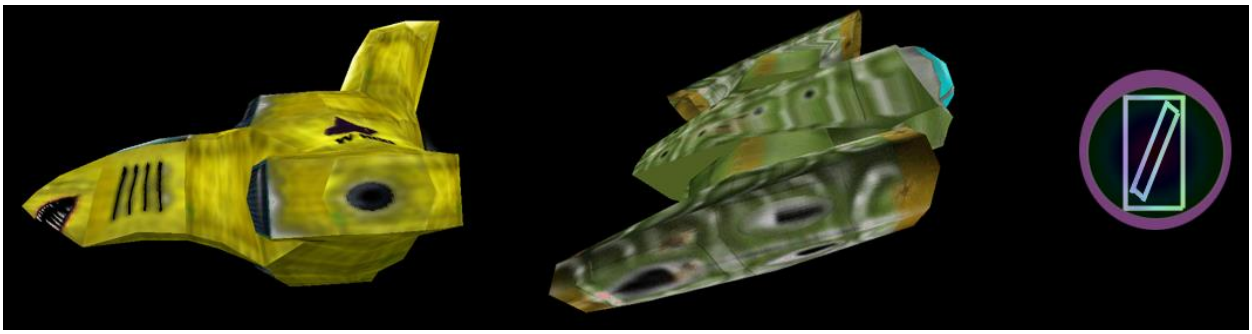
Based off of deductive research, we will assume that near the top of the hierarchy, objects will be much more cubical. This would mean that as the object is subdivided, the shape tends to morph towards skewed axes, in which the length, width, and height vary greatly from one another. Using this model, we maximize efficiency by initially doing sphere collision, since sphere volumes for cubical geometry is only a factor of about 4 less efficient than for BBs (seen in the above table), but uses CD which is at least an order of magnitude faster (more than 4 times faster). For the complicated BV CD down the hierarchy, the limit of the ratio of sphere volume inefficiency to the speed of OBB CD reaches one, and since we are trying to do as few CD tests as possible at this level, we cast the BV nodes to OBBs and use OBB CD, getting the most out of the dynamic sphere/OBB BV tree.

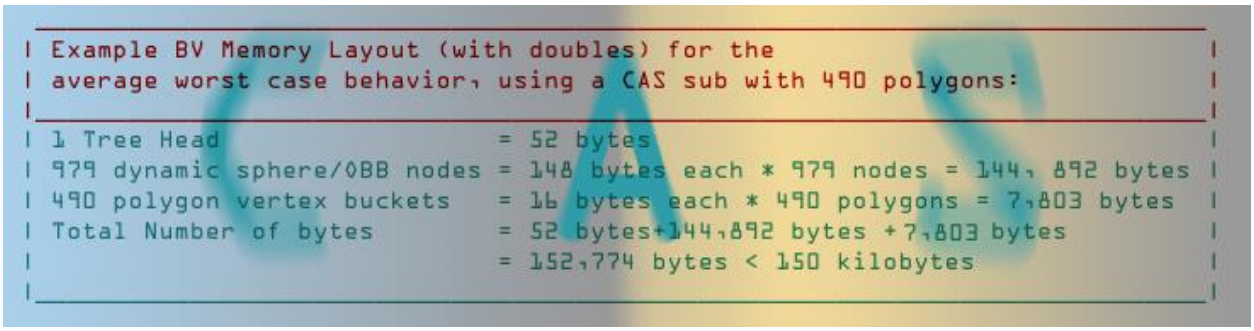
////////////////////////////////// Foot Notes //////////////////////////////////

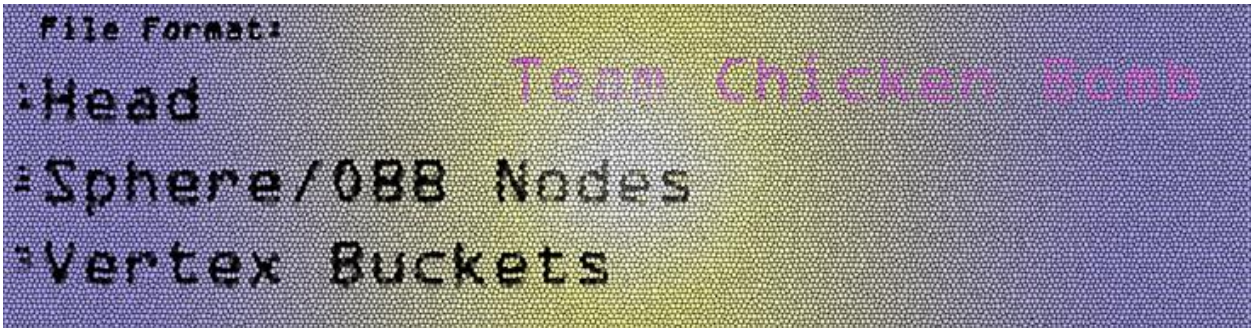
The traverse down the BV hierarchy uses a global stack big enough for the worst-case collision numbers (~n^2, about 250,000 node pointers in implementation). Since the traverse is depth first, the stack is the optimal choice. The traverse is iterative, completely bypassing the need for compiler optimization. Traversing with CD should take no more than O (n*lg (n)) time for an entire hierarchy. When minimum accuracy is required, but a point of collision has to be returned, that point will be approximated either as the point of collision between moving spheres/boxes, or between a ray and sphere/box.

//

The memory for each hierarchy ranges from 50kb to 300kb, with 100kb on the average. The memory layout for one example hierarchy is presented below, while the memory map for all the hierarchies typically encountered during a game is presented in the main TDD memory map. The file format for the BV hierarchy tree immediately follows. Closing remarks and a short bibliography conclude this TDD section.







After such a strenuous and detailed breakdown of the method of CD used in CAS, it should be clear that our method is superior to all others. Paling in comparison, all academic scholars and game companies could learn a thing or two from this revolutionary new technology. Being faster and more accurate than any current ID or BV representation in the world, we will continue to break barriers and shred the competing 3D community.

We are Team Chicken Bomb.

Bibliography:

[OBBTree] S. Gottschalk, M. Lin, and D. Manocha. “OBBTree: A hierarchical Structure for rapid interference detection,” Proc. Siggraph 96. ACM Press, 1996. has contributed a great deal to the discussion of OBBs in terms of accuracy and speed of execution.

[Jeff] When Two Hearts Collide: Axis-Aligned Bounding Boxes by Jeff Lander [02.03.00] at www.gamasutra.com Most discussions of collision detection for real-time game applications begin with bounding spheres and bounding boxes. While fast, bounding spheres don't generally give the best approximation of an object's extent. Jeff Lander demonstrates the use of axis-aligned bounding boxes for 3D collision detection.