T-Collide: A Temporal, Real-Time Collision Detection Technique for Bounded Objects

Erin J. Hastings, Jaruwan Mesit, Ratan K. Guha
College of Engineering and Computer Science, University of Central Florida
4000 Central Florida Blvd. Orlando, FL 32816
hasting@cs.ucf.edu, jmesit@cs.ucf.edu, guha@cs.ucf.edu

Abstract

This paper presents T-Collide, a fast, low memory-overhead, low execution-cost, time-based collision detection scheme. It is intended for real-time systems such as games or simulations to optimize collision detection between large numbers of mobile objects. Nearly all aspects of T-Collide are fully customizable to application specifics or implementer preference. T-Collide is based upon Spatial Subdivision, Bounding Volumes, Spatial Hashing, Line Raster Algorithms, and Continuous, or Time-Based Collision.

Keywords

temporal, continuous, real-time collision detection, uniform spatial subdivision, spatial hashing, bounding volumes, T-Collide

Introduction

T-Collide is a collision detection scheme for mobile objects in real time systems and is especially suited to computer simulations or games. It is based upon:

- Spatial Subdivision: A "Grid" divides the world into smaller areas.
- Bounding Volumes: Each complex object has a simple bounding volume.
- Spatial Hashing: A function that determines where an object is in the Grid.
- Line Raster Algorithms: When an object passes multiple grid cells in the same frame, grid location detection becomes similar to a line raster problem.
- Collision over Time: Collision is a function of position and time, not just position. The "T" signifies temporal collision.

T-Collide requires little memory overhead and virtually all aspects of the algorithm are customizable to application or implementer preference. Use of the T-Collide algorithm imparts no arbitrary restrictions on application parameters such as object size or movement rate per frame. Certain parameter ranges are however, more optimal than others.

Related Work

First, "required reading" regarding collision detection includes: the Lin-Canny closest features algorithm (Lin 1992), V-Clip (Mirtich 1998), I-COLLIDE (Cohen 1994), OBB-Trees (Gottshalk 1996), Q-Collide (Chung 1996), and QuickCD (Klosowski 1998). Another interesting approach to real-time collision is by Monte-Carlo Method (Guy 2004). Some excellent resources on general collision detection with an emphasis on games are: (Blow 1997), (Bobic 2000), (Dopterchouk 2000), (Gomez 1999), (Gross 2002), Heuvel 2002), (Lander 1999), (Nettle 2000), and (Policarpo 2001). The recent endeavor most related to this one is (Gross 2002), which also utilizes spatial hashing. The hashing methods differ considerably however, and it does not consider collision over time.

Algorithm Overview

In order to implement T-Collide, the following are required:

- The Grid
- Bounding Volumes for all collision objects
- A Hash Function for each bounding volume type
- A Hash Table

First we will discuss assumptions regarding application setup. Then we will cover specifics of the Grid, Bounding Volumes, and Hash Function, and Hash Table in turn, followed by a high level description of the T-Collide algorithm itself. Finally we will analyze the algorithm's time/space complexity, present simulation results, and conclude the paper.

Assumptions or Restrictions

Regarding general application setup, the following is assumed:

- Animation involves a typical update/draw loop.
- Collision detection is performed every frame.
- Updates are based on some TIME_ELAPSED variable representing time elapsed since the previous frame.
- Object-environment collision is performed prior to object-object collision.
- Object position at the beginning and end of each frame is saved - position_initial and position_final respectively.
- Objects and their bounding volumes are somewhat smaller than grid cells. As we will see, grid cells may be any arbitrary size, but being larger than the bounded objects increases performance.

The Grid

The grid divides the entire scene into distinct cells (uniform spatial subdivision). The grid may be 2D or 3D. A 2D grid is faster and is a better choice for applications where many objects are spread over a wide area, but rarely above each other. For example – flight simulations, naval simulations, "top down" real-time strategy games, or space shooters with many units.

The grid requires no explicit storage of cells or content. Only a single variable is required: CELL_SIZE. Note that CELL_SIZE should be significantly larger than most objects in the scene to reduce chances of objects spanning multiple cells. While a single object spanning multiple cells is acceptable (for example, objects on a cell boundary) it does slow the hashing somewhat. Thus CELL_SIZE will affect performance. Figure 1 depicts a grid.

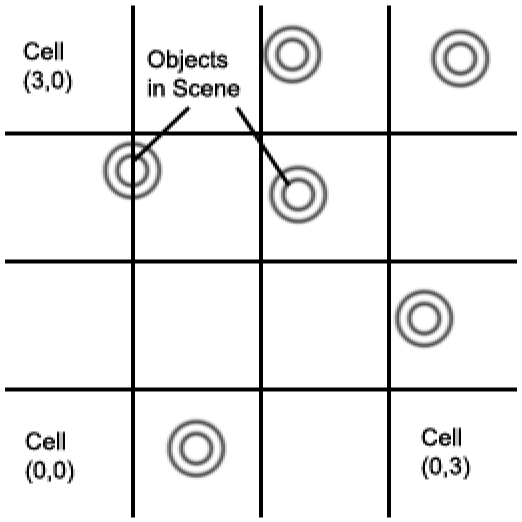


Figure 1 – The Grid

In a 2D grid, each cell is uniquely represented by two integers generated by the hash function. A 3D grid requires three integers. The hash table has a bucket for each cell. Each object in the scene is then hashed to a bucket.

Bounding Volumes

All complex models are surrounded by a simple bounding volume. Bounding volumes may serve as either:

- the object's collision model, where collision with the bounding volume signifies collision with the object
- a "first pass" indicator signifying possible collision with the object's complex model, where more detailed collision detection is performed later

In the former case, detailed collision is not required and the object's bounding volume is the object's collision volume. The latter case is obviously required where complex hit detection is required and the bounding volume serves to signify "possible collision". Either method can be employed.

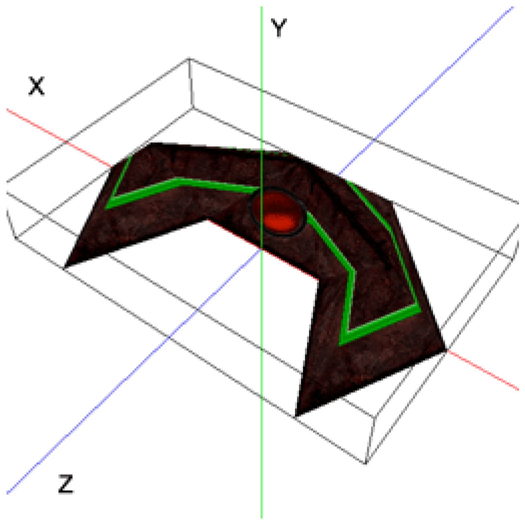


Figure 2 – Object within Axis-Aligned Bounding Box

Any bounding volume may be employed in T-Collide, but each will have a slightly different hash function. In the following examples we will use axis-aligned bounding boxes (AABB), Oriented-Bounding Boxes(OBB), and Spheres as bounding volumes for objects. An AABB can be represented by two points – min and max. An AABB never tilts; it is always aligned with the axis thus streamlining computation. OBBs are boxes as well, but tilt or rotate with the object. Figure 2 depicts an object enclosed within a bounding box.

The Hash Function

The hash function determines which grid cell an object is in and is run every frame on all objects. The hash function is based on:

- bounding volume type
- object's initial position (beginning of the frame)
- object's final position (end of the frame)

Note that an object may hash to more than one grid cell, such as when it crosses a cell boundary. For this reason, a hash table is kept where each hash bucket contains a list of which objects hash to that associated cell. We will discuss the specifics of the hash table later, but now let us examine a hash function.

Assume the following global variables or structures:

- <u>Cell Size</u>: the size of each grid cell
- <u>Grid Cell(x, z)</u>: structure to uniquely identify each grid cell denoted by x,z since the grid is 2D, therefore y-axis values may be ignored
- <u>Min, Max</u>: the axis aligned bounding box

Suppose each object in the scene has the following variables:

- <u>position_initial(x,y,z)</u>: position at the beginning of the frame
- <u>position_final(x,y,z)</u>: position at the end of the frame
- <u>boundingBox (min,max)</u>: the object's AABB

An object may span multiple cells if it is on a grid boundary line. A majority of the time however it will be a single cell, since grid cells should be much larger than individual objects.

Hashing a Single Point

First, how can we determine what grid cell a single point is in? One obvious choice for a hash function is to simply divide the object's position by "cellSize". Since division is slow and should be avoided, a better choice is to define a conversion factor that when multiplied by an object's position yields a unique grid cell. We can define the conversion factor as follows:

- conversion_factor = 1/cellSize;

So our hash function for a single point might look something like this.

```
gridCell hash(point p)
{
   gridCell g;
   g.x = p.x*conversion_factor;
   g.z = p.z*conversion_factor;
   return g;
}
```

Given point p the hash functions returns a unique grid cell. As an example, with a cell size of 10.0 hash(x=105.6, y=30.3, z=55.2) would return 2D grid cell (x=10, z=5).

Hashing a Bounding Box

We need to hash entire objects to the grid though, not just single points. Note that there are three cases where an object's AABB may span either: 1, 2, or 4 grid cells. Figure 3 illustrates this. To determine what cells are spanned by an object at a certain position, we may have to consider the four corners of the object's bounding box.

The function for determining which set of grid cells a bounded object spans is shown in figure 3. The function takes a bounded object as a parameter and computes what grid cells it hashes to. Notice that the function short-circuit evaluates based on the fact that: if min and max hash to the same cell, no further evaluation is required (which will usually be the case).

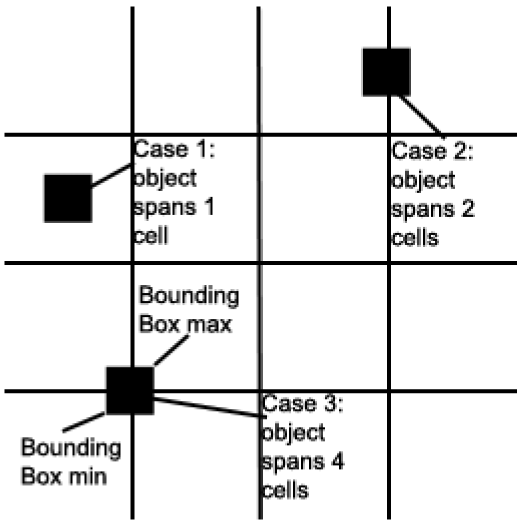

Case 1: object spans 1 cell
Case 2: object spans 2 cells
Bounding Box max
Case 3: object spans 4 cells
Bounding Box min

Figure 3: AABB Spanning Multiple Grid Cells

```
void HashGridCellAABB(Object obj)
{
   // hash object AABB min and max
   gridCell a = hash(obj.min);
   gridCell b = hash(obj.max);

   if (a=b)  // case 1
   {
       add object to hash table…
       …occupies grid cell a
   }
   else if(a.x=b.x)  // case 2
   {
       add object to hash table…
      … occupies grid cell a and b
   }
   else if (a.z=b.z) // case 2
   {
       add object to hash table…
      … occupies grid cell a and b
   }
   else  // case 3
   {
       must hash all 4 corners…
       … and add to hash table
   }
}
```

If OBBs are used, the min and max must be computed every frame due to box rotation. The max(x,y,z) point simply becomes the maximum x, y, and z values of all the OBB corners (see Figure 4). The min(x,y,z) is computed with min values, then hashing proceeds as with AABBs. Therefore OBB provide tighter bounding than AABB but slightly slower hashing.

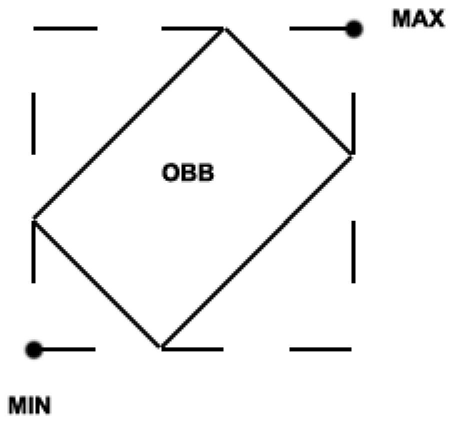


Figure 4 – Max and Min of an OBB

If bounding spheres are used then sphere center + radius values must be hashed. Namely: center.x + radius, center.x – radius, center.z + radius, and center.z – radius. Spheres will be slower to hash however since four points must always be considered (see Figure 5), whereas boxes may short-circuit evaluate if min and max hash to the same cell.

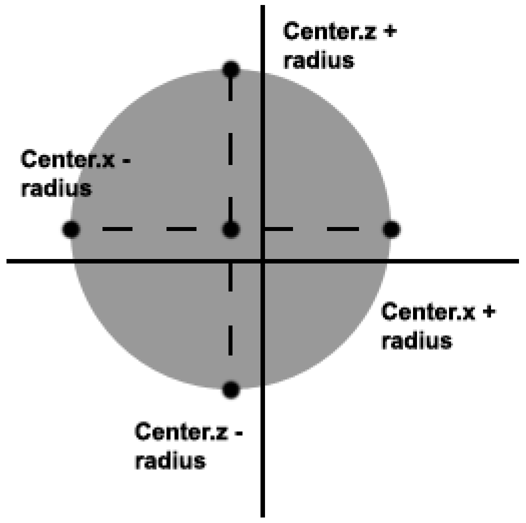


Figure 5 – Hashing a Sphere

Hashing Objects Over Time

Since collision must be determined over the course of the frame there is another factor to consider – the object's position at the start and the end of the frame. For example, suppose at time T1 Object 1 and Object 2 are in positions designated by Figure 6. Then suppose after update their new positions are as shown at time T2. If collision detection is based only on position at time T2, then this collision will go undetected. Clearly collision is a function of position and time, not just position.

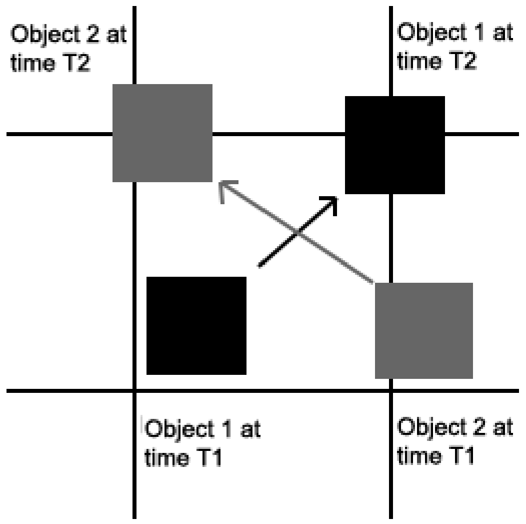


Figure 6 – Collision Over Time

Therefore, we must determine all grid cells the object has passed through during the frame. Determining which cells an object traverses is done as follows.

- Determine the initial cell the object occupies
- Determine the final cell the object occupies
- Find the cells traversed between them

The issue is somewhat similar to a line raster problem – where the endpoints of the line are the initial and final positions and the "pixels" are the cells traversed. Figure 7 illustrates this.

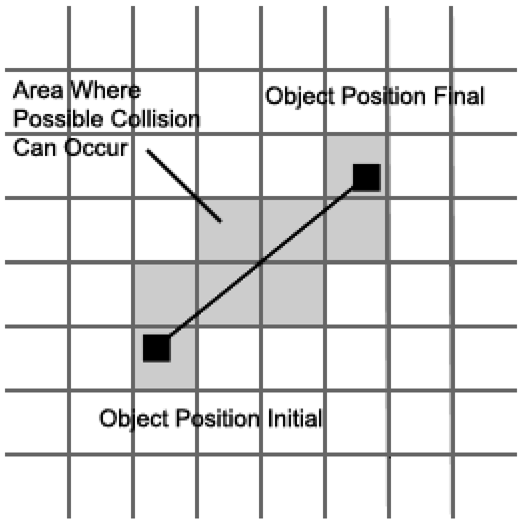


Figure 7 – Cells Traversed by an Object

There are various ways to approach this problem. One issue to note is that simply tracing a line between the two objects may yield a margin of error, since objects are obviously much wider than lines. Figure 8 illustrates this, where the object would clearly intersect the lower right cell but does not. This may be acceptable for objects that are much smaller than grid cells.

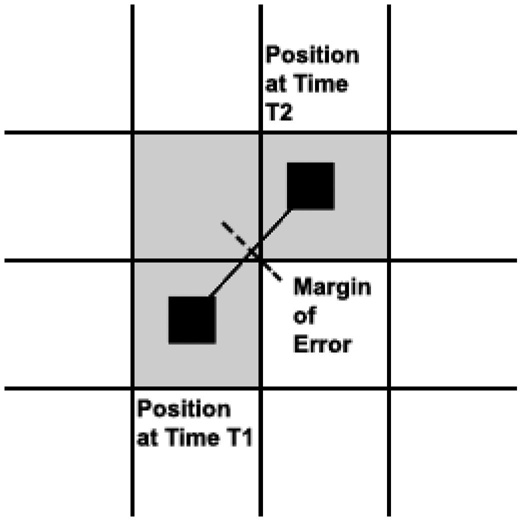


Figure 8 – Determining Cells Traversed During the Frame

For most objects, we can take a simplistic approach with no margin of error. Suppose an object hashes to cell A at the beginning of the frame and cell B at the end of the frame. We will simply "draw a box" that encompasses the cells from A to B. The algorithm works as follows where A and B are Grid Cells (x,z), A is the lower valued cell (i.e A.x < B.x and A.z < B.z), and A != B.

```
for(i=A.x; i<(B.x–A.x); i++)
  for(j=A.x; j<(B.z–A.z); j++)
  {
     Add object to the hash bucket…
     … associated with grid cell(i,j)
  }
```

This may seem a sub-optimal or "brute force" approach but consider:

- Cells are significantly larger than objects
- Time between frames is fairly small in real-time applications

From this we can conclude that an object will almost always move no more than one cell from its current position during the frame. Clearly a complex algorithm for traversing cells is not needed, and may actually slow computation somewhat. In almost all situations cell traversal over the course of the frame will be similar to one of the three cases presented in Figure 9. Thus our simple nested "for" loop is adequate.

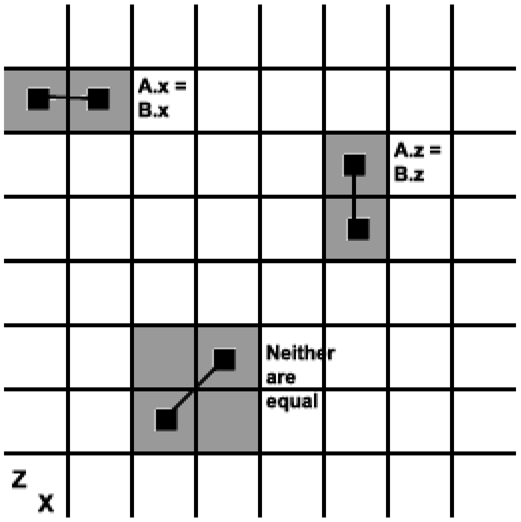


Figure 9 – Hashing Objects Crossing Cell Boundaries

There are some cases however, where tracing a line through the grid is entirely appropriate – when the object is represented by an origin and a direction vector. Bullets and other "instant hit" weapons are often implemented this way in games. A vector from the origin of the projectile out to its range can be traversed through the grid. All objects within those traversed cells are possible targets of the bullet. Note that the grid optimizes target computation in this case – if the cells are traversed in order from origin along the vector, the closest possible targets will be found first, allowing early exit.

The Hash Table

The hash table will have a bucket for every grid cell. Objects may be referenced in the hash buckets either by pointers or integers or their index in the object list. We will use the latter method in the following example. Supposed we have a list of N objects in the scene:

ObjectList[0…N]

Suppose also that we have the following variables:

GridMax = 100: the top corner of the grid
GridMin = 0: the bottom corner of the grid
Width = 100: the width of the entire grid, computed by GridMax - GridMin
CellSize = 25: the size of each cell
ConversionFactor = 0.04: computed as 1/CellSize

Figure 10 shows an example grid, 4 objects in an object list, their positions at the start and end of the frame, and the hash table. Notice that objects A and C stay in the same cell over the course of the frame, thus hashing to buckets 0 and 1 respectively. Object B crosses from cell 2 to cell 3, so it appears in hash buckets 2 and 3. Finally, object D ends up overlapping all cells. Therefore D is added to all hash buckets.

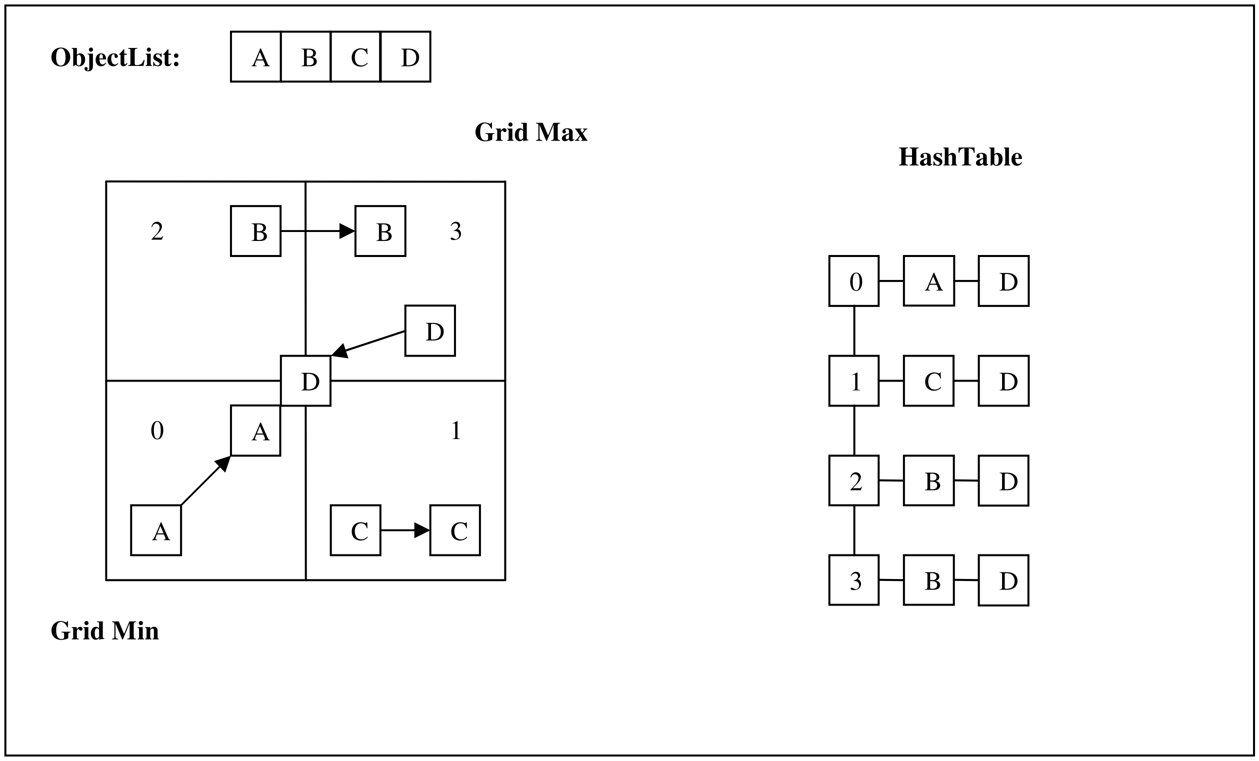


Figure 10 – Object List, Grid, and Hash Table

The T-Collide Algorithm

Given the following:

- Cell Size: size of each cell
- Grid Cell (x,z): data structure uniquely identifying each cell
- Bounded Objects: each object stores a bounding volume, a list of the grid cells it currently occupies, an initial position, and a final position
- Hash Table: hash table where each bucket is associated with a unique grid cell and the index of any object occupying that grid cell is stored

At a high level, collision detection with the T-Collide algorithm proceeds as follows:

- Hash Phase: for each object in the scene:
    - Hash object to the hash table (Hash Function)
- Collision Phase: for each bucket in the hash table:
    - Collide all objects indexed in that hash bucket, if any

The Hash Function for AABB or OBB proceeds as follows:

- A = hashed cell at beginning of frame
- B = hashed cell at end of frame
- If A == B we are done, else

```
for(i=A.x; i<(B.x–A.x); i++)
  for(j=A.x; j<(B.z–A.z); j++)
  {
    Add object to the hash bucket…
    …associated with grid cell(i,j)
  }
```

Actual collision and response will vary greatly from application to application. Some suggested resources that consider collision over time: (Dopterchouk 2000) and (Heuvel 2002) for spheres, (Lander 1999) and (Policarpo 2001) for AABBs, and (Blow 1997), (Bobic 2000), (Gomez 1999), and (Nettle 2000) for ellipsoids, rays, OBBs, or other shapes.

As a final note, one common mistake in writing collision loop algorithms that compare "all objects in the scene" is to proceed as follows:

```
for(i=0; i< num_objects; i++)
   for(j=0; j<num_objects; j++)
      if(i!=j) doCollision(object[i], object[j]);
```

This can be abbreviated to the following, which avoids the check for whether or not an object is being compared to itself (i!=j) but still compares all objects:

```
for(i=0; i< num_objects-1; i++)
   for(j=i+1; j<num_objects; j++)
       doCollision(object[i], object[j]);
```

Analysis of the Algorithm

With regard to space complexity T-Collide requires only a Hash Table of integer values. The number of integers stored in the hash table will be based on N objects in the scene:

- Best case: 1 integer per object in the scene. In this case every object hashes to a single gird cell.
- Worst Case: 4 integers (4 grid cells) per object in the scene or possibly more, but this would be exceedingly rare. In this case every object in the scene would be positioned exactly on the 4-way intersection of 4 grid lines.

With regard to time complexity the hash phase breaks down as follows, assuming use of OBB or AABB for a single object bounding volume:

- Best Case: 2 scalar multiplies (to hash min and max or an object over time) and 1 integer comparison (if A==B). This case will be the most common where each object hashes to only 1 grid cell.
- Worst Case: 3 scalar multiplies (hash 3 points) and 2 integer comparisons. Also, possibly a small nested "for" loop of integer comparisons for fast moving objects that are "traced" though the grid (bullets an example mentioned previously) As noted above, this case will be rare.

With regard to time complexity, analysis of the Collision Phase breaks down as follows supposing N objects are in the scene:

- Best Case: O(1). In this case every object hashes to a different grid cell. No collision will be performed at all.
- Worst Case: O(N^2). In this case, every object in the scene is in the same grid cell.

Obviously, performance will be somewhere between these extremes depending on object distribution in the scene. Experimental results in the next section show very substantial performance increases for collision of randomly distributed objects.

Implementation and Experimental Results

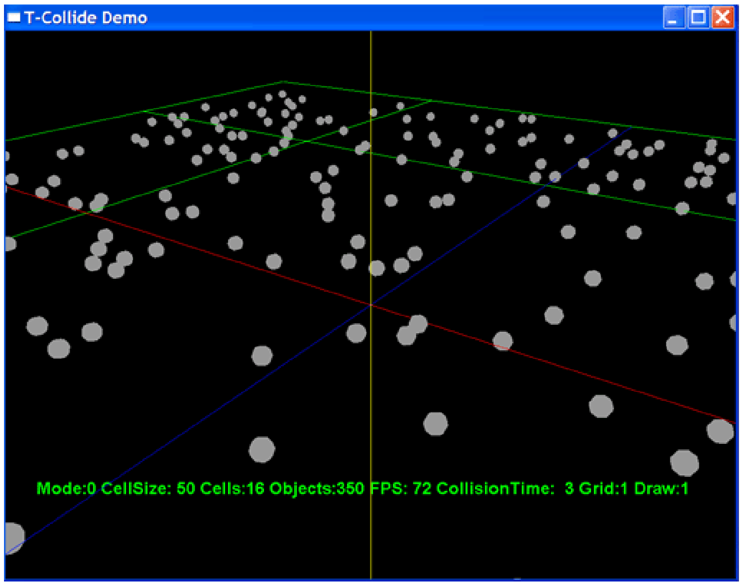
A test application was coded in C++, a screenshot of which is shown in Figure 11. OpenGL was used for graphical output and the DirectX 9 math library was used for computation.

The demo program allows for objects to be added or removed from the scene and the grid cell size to be manipulated on the fly. It also allows comparison of T-Collide and un-optimized collision by swapping between the two modes. FPS is measured using milliseconds and the collision phase is timed separately from the rest of the program using processor ticks (C/C++ clock_t type from time.h).

The hash table was implemented as a 2D dynamic array of integers using the C++ <vector> class. C++ vectors are extremely fast to access, but loading of the hash table (which must be done every frame) could likely be sped up by implementing a "true" hash table.

In the simulation each object has a bounding sphere. Collision between all objects is calculated. If a collision occurs objects are "bounced" off each other based on their velocity vectors. Thus the simulation is measuring bounding sphere collision over time and a simple collision response.

The test machine was an Intel Pentium 4, 800Mhz FSB, with 1Gig RAM, and a Radeon 9800 Pro video card. The results are shown below which compare time to compute collision for all objects in the scene in an un-optimized manner vs. T-Collide with varied cell sizes. The cell sizes are given in absolute numbers and more importantly, as a percentage of the width of the entire scene. The width of the entire grid in the demo was 400 units. Objects had a radius of 1 unit.



```
Mode:0 CellSize: 50 Cells:16 Objects:350 FPS: 72 CollisionTime:  3 Grid:1 Draw:1
```

Figure 11 – Screenshot of the Simulation

Performance was measured in clock ticks, so we provide this very rough approximation of clock ticks to frames per second. Note that this is likely processor dependent. Overall, a clock_t measurement of approximately 18 or less resulted in greater than 60 frames per second. A clock_t measurement of greater than approximately 48 yielded a frame rate of less than 20 – which is unacceptable for real time applications.

The results show that a grid cell size of approximately 1.5% – 2.5% of the grid width is optimal for a large numbers of objects (greater than 20,000). For 5000 objects or less grid size did not have much impact on performance. For less than 500 objects optimization is not required. Clearly, collision detection and response for more than 20,000 objects is feasible at over 60 frames per second. Collision for 50,000 objects was possible around 30 times per second.

Total Clock Ticks for Collision Computation	Application FPS
18	60
22	50
28	40
34	30
48	20

Figure 12 – Clock ticks and frames per second. Results showed that if the collision computation was completed in approximately 18 ticks or less, the application would run at greater than 60 frames per second. A collision computation time of 48 ticks or greater resulted in less than 20 frames per second – which unacceptable for real time applications.

	Un-Optimized	Cell Size 5	Cell Size 10	Cell Size 25	Cell Size 50	Cell Size 100
Objects		**1.25%**	**2.50%**	**6.25%**	**12.50%**	**25%**
500	2	1	1	1	1	1
1000	7	1	1	1	1	1
2500	42	1	1	1	1	1
5000	170	3	2	2	5	16
10000	~	4	5	7	17	63
20000	~	11	12	22	78	~
50000	~	35	41	153		~

Figure 13 – Comparison of un-optimized collision vs. T-Collide using differing cell size. Cell size is given as an absolute value, and as a percentage of the width of the entire area of the scene. "Objects" is the number of collision objects in the scene. Results are the number of ticks needed to calculate collision and response for all objects. For example, using T-Collide with a cell size of 10 units (which is 2.5% of the test scene width) collision and response was calculated for 20,000 bounded objects in approximately 12 ticks (well over 60 times per second).

Summary

What T-Collide does provide with respect to performance and output is:

- Low Memory Cost – T-Collide requires no additional complex storage structures to be added to an application. An integer hash table is all that is required. This makes T-Collide suitable for applications deployed on limited resource clients.
- Low Execution Cost – This is obviously required for any real-time application. A majority of object-object collisions can be resolved with the comparison of a few integers.
- Real Time Collision Detection of Bounded Objects: T-Collide is meant to be fast, and therefore ideally suited for real-time applications. It can, of course, be used in non-real time applications as well with little modification.
- Flexibility: All aspects of T-Collide are fully customizable – object bounding volumes, Grid cell size, level of detail, or the hash functions. Experimentation with variables is encouraged since they will significantly affect performance.
- Ease of Implementation: T-Collide is designed to be simple enough to add to any existing application with and provide good results with as little hassle as possible.

What T-Collide does not provide with respect to performance and output is:

- Perfect Physical Accuracy: As with most real-time graphics algorithms, the objective of T-Collide is not to model real world physics perfectly. Its objective is to provide "good" or fairly realistic results very quickly.
- Perfect Optimization: T-Collide may not necessarily be optimal in some cases. However, its simplicity provides an excellent effort/results ratio, and its low memory cost and speed provide an excellent overhead/results ratio.
- Object-Environment Collision: T-Collide does not provide object-environment collision which is better left to rendering algorithms such as BSP-trees, oct-trees, quad-trees, etc… that are optimized for huge amounts of static data. T-Collide is for collision detection between many highly mobile objects.

Future Work

Possibilities for future investigation include: the application of spatial hashing to the optimization of terrain collision, terrain rendering, particle systems, AI routines, and flexible models.

References

Blow, J. 1997. "Practical Collision Detection". Proceedings of the Game Developers Conference 1997.

Bobic, N. 2000. "Advanced Collision Detection Techniques". GamDev.net feature article.

Chung, K. 1996. "Quick Collision Detection of Polytopes in Virtual Environments". ACM Symposium on Virtual Reality Software and Technology. July 1996.

Cohen, et. al. 1995. "I-Collide: An Interactive and Exact Collision Detection System for Large Scale Environments" ACM Interactive 3D Graphics Conference 1995.

Dopterchouk, O. 2000. "Simple Bounding-Sphere Collision Detection". GamDev.net feature article.

Eberly, David. 2001. *3D Game Engine Design: A Practical Approach to Real-Time Graphics*. San Diego: Academic Press.

Gomez, M. 1999. "Simple Intersection Tests for Games". GamDev.net feature article.

Gottshalk, S. et. al. 1996. "OBB-Tree: A Heirarchical Structure for Rapid Interference Detection". Proceeding of the ACM SIGGRAPH 1996.

Gross, M. et. al. 2002. "Optimized Spatial Hashing for Collision Detection of Deformable Models". *Vision, Modeling, and Visualization 2003*. (Munich, Germany, Nov. 19-21).

Guy, S. and Debunne, G. 2004. "Monte-Carlo Collision Detection". PHD thesis paper.

Heuvel, J. and Jackson, M. 2002. "Pool Hall Lessons: Fast, Accurate Collision Detection between Circles or Spheres". GamDev.net feature article.

Klosowski, J. 1998. "Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments".

Lander, J. 1999. "When Two Hearts Collide: Axis Aligned Bounding Boxes". Game Developer Magazine. February, 1999.

Lin, M. 1992. "Efficient Collision Detection for Animation and Robotics". PHD thesis paper.

Mirtich, B. 1998. "V-Clip: Fast and Robust Polyhedral Collision Detection". ACM Transactions on Graphics. July, 1998.

Nettle, P. 2000. "General Collision Detection for Games Using Ellipsoids". GamDev.net feature article.

Policarpo, F. and Watt, A. 2001. "Real Time Collision Detection and Response with AABB". SIBGRAPI 2001.