Optimization of Large-Scale, Real-Time Simulations by Spatial Hashing

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

Keywords: spatial hashing, collision detection, frustum culling, picking, AI

Abstract

As simulations grow in scale, optimization techniques become virtually required to provide real-time response. In this paper we will discuss how spatial hashing can be utilized to optimize many aspects of large-scale simulations. Spatial hashing is a technique in which objects in a 2D or 3D domain space are projected into a 1D hash table allowing for very fast queries on objects in the domain space. Previous research has shown spatial hashing to be an effective optimization technique for collision detection. We propose several extensions of the technique in order to simultaneously optimize nearly all aspects of simulations including: 1) mobile object collision, 2) object-terrain collision, 3) object and terrain rendering, 4) object interaction, decision, or AI routines, and 5) picking. The results of a simulation are presented where visibility determination, collision and response, and an AI routine is calculated in real-time for over 30,000 mobiles objects on a typical desktop PC.

Introduction

As the number objects in a simulation increases there becomes a growing need for optimization in order to provide real-time response. The main areas where performance becomes unacceptably low for large numbers of objects are rendering, collision, and decisions or AI routines. With regard to rendering, objects not in view of the camera must quickly be discarded from consideration to maintain an acceptable frame rate. With regard to collision, it is generally an N^2 routine where all objects must be compared to all other objects in the scene. Due to the exponential nature of collision, application performance can quickly degrade as number of objects in the simulation increases. With regard to AI or autonomous agent decisions, they can often be exponential as well. For example, consider any application where objects must react to other objects in the vicinity. Clearly every object must be aware of the distance to every other object. In this paper we propose several methods to concurrently optimize collision, rendering, and AI routines in simulations based on the following established techniques:

- Spatial Hashing: where objects in 2D or 3D space are projected into a 1D hash table allowing especially fast location and proximity detection queries
- Bounding Volumes: where complex objects are encased in simple volumes for fast location and collision computation
- Position Over Time: where the position of an object is not only considered each frame (discrete), but over the course of each frame (continuous)

Specifically, using spatial hashing-based methods we will address the implementation and optimization of:

- Object-object collision: collision between mobile objects
- Object-terrain collision: traversal of terrain by the mobile objects
- Rendering: use of the grid and hash table to quickly discard objects and terrain that are not in view of the camera and thus need not be rendered (visibility determination)
- Picking and Object Selection: a type of collision detection where one or more objects in the scene are selected by users or AI entities
- AI decisions: use of the fast location and proximity detection provided by hashing to speed up AI decisions

All of the above can be simultaneously implemented or optimized with little additional memory overhead using the proposed techniques. Discussion will proceed in the following manner. First, we will examine related research, discuss any restrictions posed upon the application setup, and discuss the basics of spatial hashing and bounding volumes. Then, we will look at the details of implementing a spatial hashing based scheme to optimize collision, rendering, picking, and AI routines. Finally, we will conclude with the results of a C++/OpenGL simulation.

Related Work

Various forms of spatial hashing have been used for: real-time collision detection for simulations or games with large numbers of mobile objects [7], collision of flexible or deformable models [3] [4] [6], collision for dense mesh animations [9], penetration depth and deformable model collision response [5]. Spatial hashing methods have also been used outside the graphics and simulations area in several ways including: nearest-neighbor detection in spatial databases [12], spatial hash-joins in relational databases [10], and range-monitoring queries on mobile, real-world objects [8].

To the best of our knowledge there has been no widely published research on the application of spatial hashing to concurrently optimize collision, rendering, picking, and AI routines. Tree-based techniques do exist that may be used for both collision and rendering optimization (for example BSP trees [13]) however they may perform poorly for scenes with thousands of mobile objects since the tree must be rebuilt for every frame of animation. Rebuilding a tree every frame is at best O(NlogN) for N objects whereas rebuilding a hash table is O(N) by application of a simple hash function to each object. Tree-based solutions are further complicated by the need to insure a balanced tree. These deficiencies have been overcome by self-adjusting trees [11] where the tree is only partially updated any given frame and automatically kept balanced. Self-adjusting trees have shown to be sufficiently fast for collision detection in real-time applications with a substantial number of moving objects. However, we wish to simultaneously optimize collision, rendering, and AI routines with a single data structure.

Assumptions and Restrictions

We assume a typical real-time application setup with an update/draw loop. Every frame all objects are updated and then rendered. The only restriction posed on the simulation setup by our spatial hashing method is that most objects should be somewhat smaller than the grid cells that subdivide the scene. This not absolutely required but as we will see, it makes hashing more efficient.

Spatial Hashing Overview

Spatial hashing is a process by which a 3D or 2D domain space is projected into a 1D hash table. To implement spatial hashing at least three things are required.

- a 2D or 3D grid
- a hash function
- a hash table

First, the entire domain space is subdivided by a grid (uniform spatial subdivision) which may be 2D or 3D. The grid can be defined by three variables.

- float cell size: the size of each cell
- float min, max: two points that "anchor" the grid in the domain space

The hash function takes any given 2D or 3D positional data and returns a unique grid cell that corresponds to a 1D bucket in the hash table. Objects are hashed periodically (every frame for real-time applications usually) and their locations can then be quickly queried in the hash table. Spatial hashing can be implemented in a number of ways, but the following method is presented as an example.

Figure 1 shows a 2D grid over a domain space, 10 mobile objects, a 16 bucket hash table, and an object index (objects lettered A- J) where each object's current bucket in the hash table is indexed. We define the following variables:

- float min = 0
- float max = 100
- int cell size = 25
- int width = (max-min)/cell size = 4
- int number of buckets = width^2

The spheres represent mobile objects wandering through the grid. Each cell is 25 units (cell size) and the entire grid is 100 units across (max - min). Since the width is 4 ((max-min)/cell size), a 16 bucket hash table (width^2) is required. As mobile objects wander the grid space, they hash their position (x,y) every frame using the formula:

```
int grid_cell = ( floor(x/cell size) ) +
                ( floor(y/cell size) )*width
```

This formula translates a 2D object location into a single integer – the unique grid cell that the object occupies. The hash function can be made considerably faster by removing division with a new variable "conversion factor" which is set as 1/cell size. We can also remove the floor function by allowing type coercion to truncate our positional data (floating point) to a hash bucket (integer). So the modified hash function will look like this:

```
int grid_cell =  x*conversion factor +  y*conversion
                 factor*width
```

There are two ways to update the hash table. If all objects are re-hashed every frame (usually the case in animated simulations or games) the contents of every bucket can simply be deleted before update. If objects are only updated infrequently, the old hash-data for the object must be deleted before the new hash-data is inserted. This can be easily determined using the object index.

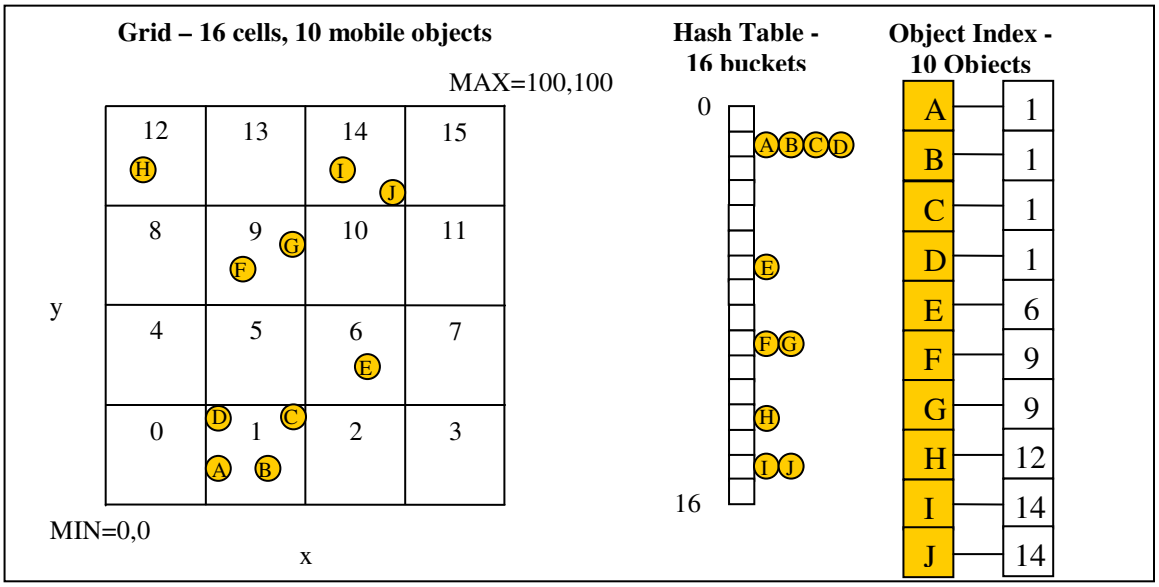

Figure 1 – An example of mobile objects in a grid, a hash table, and the object index.

Now that the hash table and index are built we can make several types of queries on it:

- Cell Query (Which objects are in cell X?): O(1) by direct access of hash bucket X
- Object Query (In which cell is mobile object A located?): O(1) by direct access of the object-index
- Proximity Query (Which objects are near object A?): the range of buckets is computed, then an O(1) cell query performed on each bucket

As we will see, these query types can be used to effectively optimize collision, rendering, and AI routines.

Bounding Volumes

All complex models we consider shall be surrounded by a simple bounding volume to greatly reduce computation required for both hashing and collision. 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

Essentially any bounding volume can be used in spatial hashing, but in the remaining discussion we will use axis-aligned bounding boxes (AABB). An AABB can be represented by two points – min and max. An AABB never tilts despite movement of the enclosed object; it is always aligned with the axis thus streamlining computation.

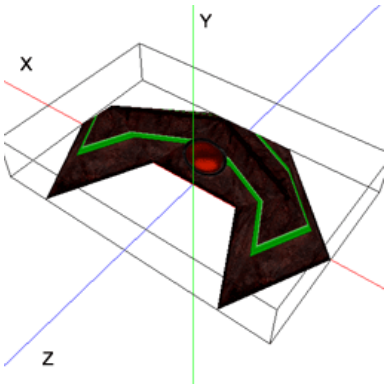

Figure 2 – An object within an Axis-Aligned Bounding Box (AABB).

Hashing an Object

Previously we discussed the hashing of a single point; however we wish to quickly hash entire objects. Hashing is based on bounding boxes and therefore objects may hash to multiple cells in the grid. An AABB may span 1, 2 or 4 cells as illustrated in figure 3. To determine what cells are spanned by an object at a certain position, the four corners of the AABB are considered. Notice that the hash function for AABB can be short-circuit evaluated based on the fact that: if min and max hash to the same cell, no further evaluation is required. As mentioned previously, it is advantageous for objects to be smaller than grid cells since the hash function proceeds faster, and each individual object hashes to fewer cells. However, if cells are too large more objects must be considered for potential collision. Thus selection of a proper grid cell size is a matter of experimentation based on number of objects, size of objects, and expected distribution of the objects in the scene. A more detailed look at the hashing of AABB and other object types (oriented bounding boxes and spheres) is presented in [7].

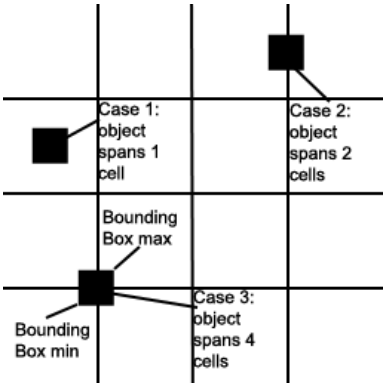

Figure 3 - An AABB will span multiple cells in 3 distinct cases. These cases may be used to quickly hash the object.

Hashing Objects Over Time

Since collision for mobile objects 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 4. 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.

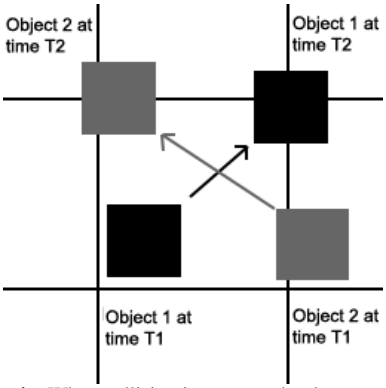

Figure 4 – When collision is computed only every frame some collisions may "miss".

Therefore in applications where precise collision over time is required, we must determine all grid cells the object has passed through during the frame. For small or very fast moving objects determining which cells are traversed is done as follows.

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

If objects are small compared to grid cells, this issue may be 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 5 illustrates this. For most objects we can take more simplistic (and faster) approach however. 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.

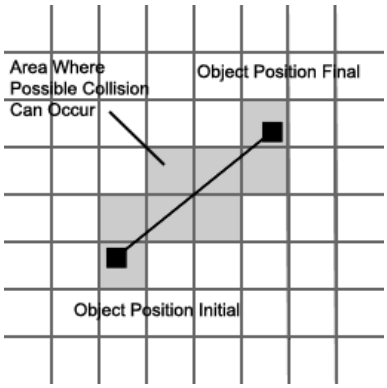

Figure 5 – Finding the cells traversed by small, fast moving objects is similar to rastering a line.

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 hash 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 6. Thus our simple nested "for" loop is adequate. The above discussion suggests at least two types of hashing based on object behavior.

- small, fast collision objects: bullets for example, will be line traced through the grid
- normal collision objects: simple nested for loop to "draw a box"

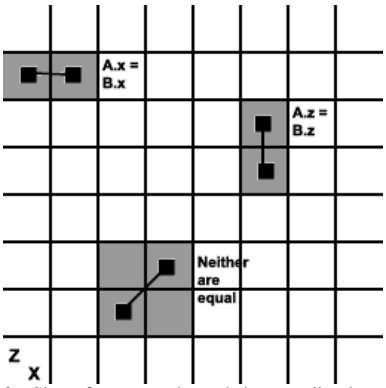

Figure 6 – Since frame rate in real-time applications is high; generally objects will move no more than 1 cell per frame. This fact can be used to simplify cell occupancy determination. The positions A and B are an objects respective position at the start and the end of a frame of animation.

Collision Detection

Optimized collision proceeds by only colliding objects that hash to the same bucket. In this manner, objects that can possibly collide (those in the same cell) are quickly found with an O(1) cell query. There are 2 phases to collision:

- Hash Phase: all mobile objects are hashed (static objects are only hashed once at startup)
- Collision Phase: for every hash bucket, collide the objects in that bucket

As noted above the hash function automatically accounts for objects that cross cell boundaries. Such objects will be referenced in multiple hash buckets. It is possible that further optimization could be obtained by designating a third class of "seldom moving" objects. Such objects that only rarely move might only be hashed after a movement. Note that static objects need only be hashed once at application startup. A hashing based terrain traversal method is described below in the simulations results section. Exact collision formulas are bounding volume dependent and will vary across applications. Note also that collision over time is optional and many applications may not require it. Further discussion of spatial hashing with regard to mobile object collision can be found in [3] and [7]. Discussion of the mathematics of mobile object collision over time can be found in [2] and [14].

Picking

Picking is a type of collision detection where user input (usually a mouse-controlled pointer) selects objects in the scene. Every object in the scene can possibly be picked but by using the hash table we can quickly discount a majority of objects. Picked objects can be found by:

- projecting a ray through the screen where the user clicked
- in a 3D grid, tracing the ray through the grid
- in a 2D grid intersecting the ray with grid plane, then hashing the intersection point
- objects in the hash buckets corresponding to intersected grid cells are candidates for picking

Additionally, allowing the user to "click and drag" a box on screen (similar to figure 7) is a common method of allowing selection of objects and can efficiently implemented by: 1) hashing the max point of the selection, 2) hashing the min point of the selection, and 3) finding the cells between them as described previously.

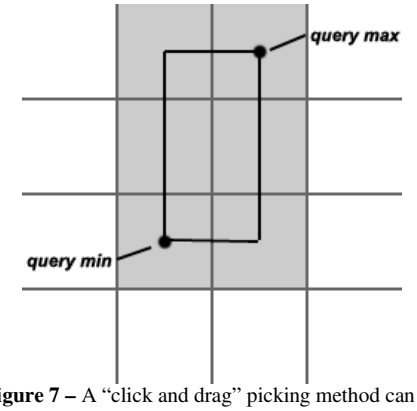

Figure 7 – A "click and drag" picking method can be implemented very efficiently. Only objects within the shaded region are possibly in the selection region.

Rendering

Visibility determination is the process of finding objects that are in view of the camera and rendering only those objects. Visibility determination with the hash table proceeds as follows.

- find which hash cells are in view of the camera
- only render objects in those cells

We will look at two different ways to perform visibility determination using the hash table. The two methods are based on application type which affects the typical "view" or behavior of the camera during rendering.

- Top-down camera: where the camera is locked in a view looking down at the scene
- Free Camera: where the camera rotates freely

A top-down camera scheme is commonly used in simulations, 2D applications, and many real-time strategy (RTS) games where the camera is above the domain space and is fixed looking down. A 2D grid naturally lends itself to this type of application since mobile objects may wander over a 3D terrain, but in general are spread out in two dimensions. Finding objects in view of a top-down camera proceeds as a somewhat simplified form of frustum culling. First we define:

- frustum max point(x,y): the point found by projecting a line from the camera through the top-right corner of the screen, intersecting the line with the grid plane, and then hashing that point
- frustum min point(x,y): the point found by projecting a line from the camera through the bottom-left corner of the screen, itersecting the line with the grid plane, and then hashing that point

The objects in cells between those that contain frustum max and frustum min are objects in view of the camera (see figure 8). Note this method only works when the camera is locked above the grid and special care must be taken when one of the points projects off the grid.

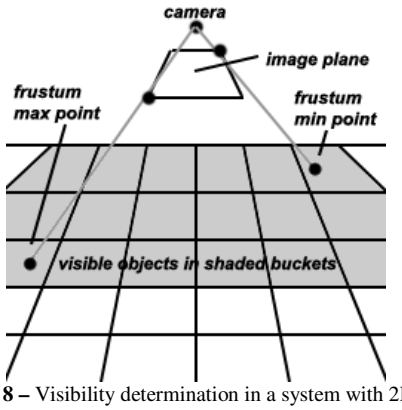

Figure 8 – Visibility determination in a system with 2D grid and top-down camera view.

After determining frustum max point and frustum min point, the cells between can be determined in the following manner. First we define:

```
float conversion_factor = 1/cell_size
fint width = (max-min)/cell_size

int draw_bucket

int minx = frustum_min_point.x * conversion_factor
int minz = frustum_min_point.z * conversion_factor

int maxx = frustum_max_point.x * conversion_factor
int maxz = frustum_max_point.z * conversion_factor
```

Conversion factor and width have already been defined for our hash function. Draw_bucket will hold the value of any bucket whose contents are drawn on screen. Minx, maxx, minz, and maxz are used to compute a grid from the frustum min point to the frustum max point. We now compute the individual draw buckets to draw as follows:

```
for(i=minx; i<=maxx; i++)
{
    for(j=minz; j<=maxz; j++)
    {
        draw_bucket = i*width + j;

        …render contents of draw_bucket…
    }
}
```

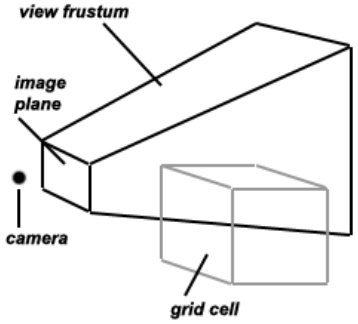

Figure 9 – Frustum culling may be performed with a 3D grid. If a given grid cell does not intersect the view frustum, objects within that grid cell are not visible to the camera.

The simulation for which results are presented later uses a technique similar to the above. In applications where the camera rotates freely "true" frustum culling is employed (see figure 9).

- mobile objects are hashed to their respective buckets
- the frustum is intersected with each hash bucket
- if a bucket does not intersect with the frustum, objects contained in that bucket are not in view of the camera

As for the grid cell vertices, they may either: 1) be computed on the fly using min, max, cell size, and an offset based on hash bucket, or 2) be stored explicitly in a 2D or 3D array. Further discussion of frustum culling via bounding boxes can be found in [1].

AI and Decision Making

AI-related routines can be implemented or optimized in several ways using the hash table.

- Proximity-based AI decisions: object proximity can be quickly estimated without explicit distance calculation
- Designated zones or areas of interest: a method of designating areas or volumes of the domain space as having certain characteristics that affect navigation or AI decisions
- Radar or sensing: where a radar-like functionality is required that provides a condensed, or alternate representation of the domain space or section of the domain space

First, any AI decision based on evaluation of objects in the vicinity can be quickly resolved using a proximity query (see figure 10). For example, suppose a decision made by an AI is affected by all other AI objects within a radius of distance. To implement such a query we can define:

- float radius: distance of the query
- int base cell: the (x,y) cell the object making the query occupies
- int offset: number of grid cells in the x,y axis around the base cell computed by radius/cell size
- int next bucket: the next bucket in the area covered by the query

We now compute each bucket in the query:

```
(for i=base-offset; i<base+offset; i++)
{
  (for j=base-offset; j<base+offset; j++)
  {
    next_bucket = i*width + j;
  }
}
```

For a 3D grid this would be computed in three dimensions. As noted previously, an O(1) query on the hash table returns the objects in each bucket computed – which are the objects within a radius r of the object.

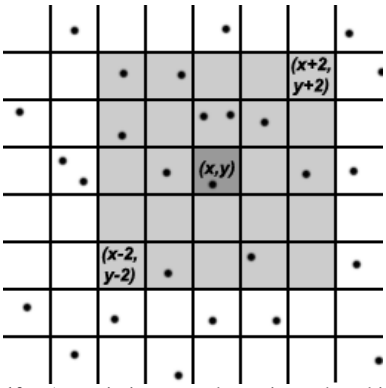

Figure 10 – A proximity query determines what objects are in the vicinity (shaded area) of a specific object. The buckets surrounding the object can be quickly found using the object's base cell, the radius of the query, and an offset.

Cell queries provide a fast way to find objects in a limited section of the domain space surrounding an object. This is ideal to implement a radar-like functionality for user or AI which is fairly common in simulations and games. Additionally, cell queries need not be limited to the granularity of the grid. More precise queries (see figure 10) can be represented by:

- an explicit query area or volume: a sphere or cube for example
- a range of cells: that the query volume hashes to

Thus any object within the range of query cells is intersected with the query volume. The query cells may be re-hashed upon movement or re-sizing of the query volume. This provides an excellent way of quickly determining objects within an area or volume. Proximity queries also allow certain areas or volumes of the domain space to be designated as special areas of interest. Such query areas may have properties that affect AI or user behavior.

Performance Analysis

Let us now analyze performance and memory requirements for the proposed methods. With regard to time complexity:

- Hashing: O(N) for each of N objects in the scene the hash function is applied
- Cell Query (Which objects are in cell X?): O(1) by direct access of hash bucket X
- Object Query (In which cell is mobile object A located?): O(1) by direct access of the object-index
- Proximity Query (Which objects are near object A?): the range of buckets is computed, then an O(1) cell query performed on each bucket

As for performance with regard to collision, visibility determination, and AI decisions, it will depend heavily on object distribution throughout the domain space. Collision performance 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.

Clearly performance will be somewhere between these extremes depending on object distribution in the scene. AI is affected by object distribution similarly to collision. With regard to speedup for rendering optimization:

- Visibility Determination: O(N) where there are N hash buckets in the table

When performing 3D frustum culling each of N cells is intersected with the frustum. With regard to system memory requirements a spatial hashing-based method requires at least:

- Hash Table: 1 to 4 integers or references per object for most bounded objects
- Object Index: 1 to 4 integers or references per object for most bounded objects, the object index is optional however and only required for certain queries

Each bounded object will typically span 1 to 4 cells. Thus a reference or integer is placed in the appropriate hash bucket for each cell it occupies.

Simulation and Results

A simulation was developed using the following technologies:

- Code: C++
- Graphics: OpenGL
- Windows Framework: nehe.gamedev.net OpenGL Application Framework
- Math Libraries: DirectX 9.0c
- Compiler: Microsoft Visual Studio.net 2003 Version 7.1.3088

The simulation consisted of mobile objects (multi-colored spheres) that randomly roam over a fixed, randomly generated terrain (triangle mesh). A screenshot of the simulation is shown in Figure 11. The hash table was implemented using a two-dimensional vector (C++ vector class).

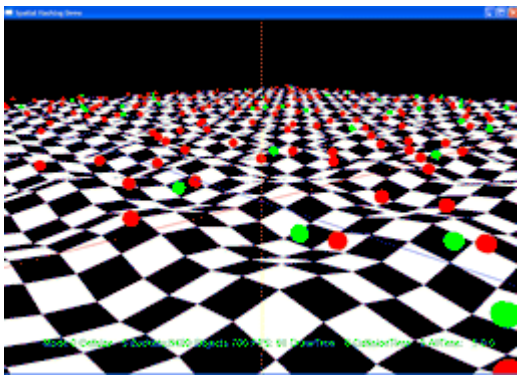

Figure 11 – The simulation of several thousand mobile objects wandering over a terrain separately times collision, rendering, and AI functions, and allows toggling of camera types and manipulation of grid variables.

The three main phases of computation were benchmarked independently:

- Collision: The collision phase consists of computing collision between all spheres in the scene, collision of all spheres to the terrain, and applying a response where colliding spheres "bounce" based on their respective impact angles.
- Rendering: In the rendering phase all spheres and terrain triangles are drawn. Visibility determination is performed with the 2D grid method discussed previously.
- AI: An AI routine is applied in this phase where each mobile object will examine the other objects within a radius. The object will count the number of different colored objects in the vicinity from a fixed set of color. The object will then change to the least occurring color.

A random terrain is generated at startup with the restriction that no terrain surface may be vertical. All terrain mesh triangles are hashed once (since they are static). The terrain traversal algorithm then proceeds as follows. Every frame each mobile object is hashed. It is then determined which triangle an object rests upon by 2D collision on the X,Z plane. This is fast and is feasible since we are assured that no triangle is at 90 degrees – thus no concave surfaces or "caves" in the terrain. The height of the object is then determined by using 1) the radius of the object's collision hull, and 2) the interpolated Y values of the triangle vertices (essentially a height map) based on the object's position.

The test machine for the simulation was a desktop PC: Intel Pentium4 3.2GHz, 1 Gig of RAM, and Radeon 9800 Pro GPU. Figures 13 through 15 display the results of the simulation. The results are presented in frames-per-second (FPS) for each phase of the algorithm. "60+" denotes that phase can be computed over 60 times per second. "<1" denotes the phase was computed less than once per second. "Objects" denotes the number of mobile objects in the simulation. "Cell size" denotes the cell size in world units of each grid cell. The percentage value denotes the size of the cell in relation to the entire domain space. Since the domain space was 400 units across, a cell size of 100 is equivalent to 25% of the domain space. Objects had a radius of 1.0. "Un-optimized" denotes measurement for the application with no optimzation whatsoever. This is provided to illustrate the magnitude of speedup from the hashing and to allow other methods to be compared with these results. The highlighted cells indicate a computation time too slow for real-time applications (under 20 FPS). The slowest phase was AI calculation and the fastest phase was visibility determination.

9. Conclusions and Future Work

Overall the simulation showed that using the spatial hashing-based techniques presented here, collision, collision-response, terrain traversal, proximity-based AI routines, and visibility determination can be computed for well over 30,000 mobile objects simultaneously at real-time frame rates on a desktop PC. The test machine was able to run the entire application (all phases, while processing user input and OpenGL drawing) with 30,000 at between 20 and 30 FPS. In order to provide faster hashing grid cells should be somewhat larger than the average object. However, the grid cells should be small compared to the size of the domain space. Smaller cells result in fewer $O(N^2)$ comparisons of all objects inside those cells. Although, smaller cell size results in larger memory usage since objects are more likely to span multiple cells. Thus there is a memory-performance tradeoff, where smaller cells (to a certain point) provide better performance but at the cost of more memory.

One possibility for future work certainly includes a type of auto-adjusting grid that changes in with regard to number of objects, distribution of objects, and average size of objects in the scene. Since the hash table is wiped every frame, changing the hash function and grid cell size at any time is allowable. Other possibilities for future research on the application of spatial hashing to the graphics and simulations area include: multi-grid hashing where several grids partition the scene at differing levels of granularity, examination of the efficiency of hashing different bounding volume types, more detailed work and benchmarks on 3D hashing, closer examination of the tradeoff between cell size, performance, and memory overhead, and hash table optimization.

Objects	Un-Optimized	Cell Size 5 1.25%	Cell Size 10 2.50%	Cell Size 25 6.25%	Cell Size 50 12.50%	Cell Size 100 25%
500	60+	60+	60+	60+	60+	60+
1000	60+	60+	60+	60+	60+	60+
2500	25	60+	60+	60+	60+	60+
5000	<1	60+	60+	60+	60+	60+
10000	<1	60+	60+	60+	60+	2
20000	<1	60+	60+	50	5	<1
50000	<1	29	23	<1	<1	<1

Figure 13 – FPS for the collision phase where collision, response, and terrain traversal are computed. With a cell size of 5, collision was computed for 20,000 objects over 60 times per second. Collision for 50,000 objects was computed around 30 times per second. Without optimization less than 3000 objects could be supported in real-time.

Objects	Un-Optimized	Cell Size 5 1.25%	Cell Size 10 2.50%	Cell Size 25 6.25%	Cell Size 50 12.50%	Cell Size 100 25%
500	60+	60+	60+	60+	60+	60+
1000	60+	60+	60+	60+	60+	60+
2500	48	60+	60+	60+	60+	60+
5000	16	60+	60+	60+	60+	60+
10000	<1	60+	60+	60+	60+	60+
20000	<1	60+	60+	60+	60	49
50000	<1	60+	60+	60+	52	40

Figure 14 – FPS for the visibility determination phase where frustum culling and OpenGL draw calls for all visible objects are made. Culling and draw calls could be performed for 50,000 objects over 60 times per second. Less than 5000 objects could be supported at real-time frame rates without optimization.

Objects	Un-Optimized	Cell Size 5 1.25%	Cell Size 10 2.50%	Cell Size 25 6.25%	Cell Size 50 12.50%	Cell Size 100 25%
500	60+	60+	60+	60+	60+	60+
1000	60+	60+	60+	60+	60+	60+
2500	35	60+	60+	60+	60+	60+
5000	<1	60+	60+	60+	60+	60+
10000	<1	60+	60+	24	33	10
20000	<1	23	28	12	<1	<1
50000	<1	8	<1	<1	<1	<!

Figure 15 – FPS for the AI phase where a proximity based AI routine for all objects is computed. AI behaved much like collision but was slightly slower. At a cell size of 5, computation for 20,000 objects could be performed approximately 50 times per second. Less than 3000 objects could be supported in real-time without optimization.

10. References

[1] Asserson, U. and T. Moller. 2000. “Optimized View Frustum Culling Algorithms for Bounding Boxes”. *Journal of Graphic Tools 2000.*

[2] Blow, J. 1997. Practical Collision Detection. *Proceedings of the Game Developers Conference 1997*.

[3] Gross M.; B. Heidelberger; M. Muller; D. Pomernats; M. Teschner. 2003. “Optimized Spatial Hashing for Collision Detection of Deformable Models”. *Vision, Modeling, and Visualization 2003*.

[4] Gross M.; B. Heidelberger; M. Muller; D. Pomernats; M. Teschner. 2004. Collision Detection for Deformable Models. *Eurographics 2004*.

[5] Gross M.; B. Heidelberger; M. Muller; D. Pomernats; M. Teschner. 2004. “Consistent Penetration Depth Estimation for Deformable Collision Response”. *Vision, Modeling, and Visualization* 2004.

[6] Guha, R.; E. Hastings; J. Mesit. 2004. “Optimized Collision Detection for Flexible Objects in a Large Environment”. *International Conference on Computer Games: Artificial Intelligence, Design, and Education 2004*.

[7] Guha, R.; E. Hastings; J. Mesit. 2004. “T-Collide: Temporal, Real-Time Collision Detection for Mobile Objects”. *International Conference on Computer Games: Artificial Intelligence, Design, and Education 2004*.

[8].Hastings, E. and R. Guha. 2005. “Real-Time Range Monitoring Queries on Heterogeneous Mobile Objects by Spatial Hashing”.

[9] Kanai, T.; and R. Kondo. 2004. “Interactive Physically Based Animation System for Dense Meshes”. *Eurographics 2004*.

[10] Lo, M. and C. Ravishankar. 1996. “Spatial Hash-Joins”. *ACM SIGMOD International Conference on Management of Data 1996*.

[11] Luque, R.; Comba, J.; Freita, C. 2005 “Broad Phase Collision Detection Using Semi-Adjusting BSP Trees”. *Proceedings of ACM Siggraph Interactive 3D Graphics and Games 2005*.

[12] Mamoulis, N.; D. Papidias; Y Tao; J Zhang. 2001. “All-Nearest-Neighbors Queries in Spatial Databases”. *IEEE Conference on Scientific and Statistical Database Management 2001*.

[13] Naylor, B. 1992. “Interactive Solid Geometry via Partitioning Trees”. *Proceedings of Graphics Interface 1992*.

[14] Policarpo, F. and A. Watt. 2001. “Real Time Collision Detection and Response with AABB”. *SIBGRAPI 2001*.