Fast Collision Detection with an N-Objects Octree
Technical Report
OSU-ACCAD-12/93-TR7

J. Edward Swan II

August 1994

Abstract

This report describes a collision detection system which uses the N-objects octree algorithm. This technique is described by Shaffer & Herb in an unpublished report [1991], and subsequently published in *IEEE Transactions on Robotics and Automation* [Shaffer & Herb 1992]. The algorithm was developed and tested with the Hook [1992] animation system. The algorithm and its implementation are described, with particular attention to the design of the octree component as an abstract data type (ADT). In addition, timing test results are reported.

Keywords

Collision detection, octree, N-objects octree, spatial data structures, dynamics simulation.

1 Introduction

1.1 The Collision Detection Problem

Cameron [1990] describes the collision detection problem succinctly: given some objects and desired motions, do the objects come into contact during the time span needed to complete the motions? The solution to this problem is useful in robotics, animation, simulation, and other problem domains. For example, Moore & Wilhelms [1988] note that with many keyframe animation systems the animator must examine the animation frame-by-frame to determine if objects are colliding. Collision detection can automate this task and alert the animator. There are also industrial applications: Shaffer & Herb [1991] describe a system which detects when the arm of the space shuttle is about to collide with objects or the walls of the shuttle's equipment bay.

Given a set of objects and desired motions, Cameron [1985] gives three general techniques for detecting collisions:

1) Divide the time necessary for the objects to complete their motions into small time intervals t_i. For each t_i, perform static intersection testing between all objects.
2) Compute the volume swept out by each object during its motion, then test the resulting 3D volumes for intersection.
3) Create a model of each object and desired motion in 4D space-time, and test the resulting 4D shapes for collision.

The method reported in this paper is based on Cameron's first approach. In general, algorithms using this technique are easy to implement, and are applicable to a wide variety of domains. The technique has a fidelity constraint, however: the distance objects move during each time interval

must be small compared to their sizes and velocities. If this constraint is not satisfied, then it is possible for two objects to pass completely through each other during a time interval.

Another problem with the first technique is that typically the spatial ordering of objects does not mirror their ordering in computer memory. Thus to determine if any objects are colliding, it is necessary to compare each object against every other object. Since this is an inherently $O(n^2)$ operation, with enough objects this comparison dominates execution time. This problem motivates the study of *spatial data structures* [Samet 1990A]. These impose a spatial structure on the objects, where the spatial extent of this structure mirrors its ordering in memory. The data structure groups those objects that are close together in space, and thus a given object is only tested against neighboring objects.

The system described in this report, hereafter referred to as *the system*, uses an N-objects octree to impose a spatial ordering on a collision detection system. The basic methods of building and maintaining the octree are given by Shaffer & Herb [1991], while Samet [1990A] gives methods for fast neighbor-finding.

1.2 Collision Response

The *dynamics* of a system govern colliding objects' behavior. The point of this work is the detection of a collision, however, and not the response to a collision. Thus, a very basic dynamics system was developed: each object is given a direction vector and a velocity; at each time step each object moves along its direction vector multiplied by its velocity. When two objects collide, their direction vectors are reversed. These simple dynamics served to develop and test the collision detection system.

1.3 The Hook Implementation Platform

The system is implemented in the C language, using Hook [1992], a track-based animation system. Hook contains both a command-line interface and a C-language programming interface; most capabilities are available through either interface. Hook contains an integer clock, which counts up from zero towards infinity. It loads objects represented as a connected mesh of polygons. Each object has a position track, which specifies a position at each clock tick. When the clock is run, each object is sequentially rendered at each position.

On SGI machines Hook uses the hardware-based GL library for rendering. This gives fast, Z-buffer quality images with a single light source and Gouraud shading [Rodgers 1985]. On Sun machines Hook uses the Xlib library to render opaque objects with hidden lines. It sorts all polygons in increasing depth order from the eye point, and then draws them back-to-front using a painter's algorithm [Rodgers 1985]. When polygons cannot be uniquely sorted in depth order, the algorithm paints them in an arbitrary order, which sometimes results in incorrect rendering (although Hook is quite useful despite this). Hook renders much faster on SGI than Sun machines.

Through its programming interface Hook provides a number of abstract data types (ADTs) which are useful for graphics programming. These include a linked list ADT, a matrix ADT, a polygon mesh ADT, a quaternion ADT, a stack ADT, and a vector ADT. Hook has additional capabilities which are not used by this project [Hook 1992].

Hook greatly facilitated the project by providing a platform for object loading, viewing, and rendering, with an easy programming interface. The author was free to concentrate on the collision detection project, without having to develop code to view or render objects.

1.4 Exhaustive Collision Detection

The system contains an exhaustive collision detection algorithm influenced by the work of Hahn [1988] and Moore & Wilhelms [1988]. Each object is tested against every other object, which is inherently $O(n^2)$ in the number of objects. The algorithm is hierarchical: to detect a collision between a pair of objects, the system calls a series of partial decision mechanisms, each with increasing computational complexity. At the top of the hierarchy is a test for divergent axis-aligned bounding boxes, which very quickly detects non-collision. At the bottom of the hierarchy, each edge of one object is tested against each polygon of the other, and vice versa. This final test is a complete decision mechanism: subject to the constraint given above (velocities are small compared to the size of objects), the final test definitely detects either collision or non-collision.

2 The N-Objects Octree Algorithm

The N-objects octree works on top of the exhaustive collision detection algorithm. A general octree is a hierarchical data structure that recursively divides a cubic volume into eight sub-volumes until a certain constraint is met. The familiar implementation is a tree with an out-degree of eight, although Samet also gives pointerless representations [1990A, 1990B]. The constraint varies and gives rise to different types of octrees. Perhaps the most common is the *region octree*: beginning with a cube that encompasses all of the objects, the octree is decomposed until each leaf node either lies completely within an object or is completely empty. In contrast, my system uses an *N-objects* constraint: the octree is decomposed until each leaf node contains no more than N objects. The N-objects octree has been used by various researchers [Shaffer & Herb 1991]; Figure 6 shows an example of an N-objects octree with N = 2.

2.1 The Octree ADT Interface

In order to increase its reusability, I implemented the octree as an abstract data type (ADT). This separates the octree from the implementation details of the current system. For example, the fact that Hook stores objects as a mesh of polygons is an implementation detail, and is not reflected in the octree ADT. The octree interface consists of two files: “octree.h”, and “neighbor.h”.

Octree Types and Operations. In order to minimize the impact on a client program’s global namespace, all of the data types and operations exported by the octree ADT are prepended with ‘oct’. The basic octree types and operations are defined in the file “octree.h”. It exports the two data types *octDirection* and *octNode* (Figure 1). An *octDirection* quantizes the possible directions in a cubical lattice to the 26 which are defined by the faces, edges, and corners of a cube; the direction names are taken from Samet [1990A]. The directions of the faces of a cube are L(eft), R(ight), D(own), U(p), B(ack) and F(ront). The directions of a cube’s edges are combinations of the two faces that are incident on each edge: LD, LU, LB, LF, RD, RU, RB, RF, DB, DF, UB, UF. And the directions

```
typedef enum octDirection {     /* Direction in the Octree sense */
   L, R, D, U, B, F,            /* Face directions */
                                /* Edge directions */
   LD, LU, LB, LF, RD, RU, RB, RF, DB, DF, UB, UF,
                                /* Vertex directions */
   LDB, LDF, LUB, LUF, RDB, RDF, RUB, RUF,
   OMEGA                        /* Undefined direction */
} octDirection;

typedef struct octNode;         /* Octree node */
```

Figure 1: two data types exported by the octree ADT.

Primary Operations:

```
N     octCreate()               Create a new root node.
void  octDispose(N)             Recursively dispose of octree from node N.
void  octClear(N, kill)         Recursively remove items from root node N.
N     octParent(N)              Return parent of N.
N     octChild(N, O)            Return O child of N.
O     octChildType(N)           Return child type of N in octParent(N).
bool  octIsSplit(N)             Is N an internal or leaf node?
void  octSplit(N)               Split a leaf node; now it has children.
void  octJoin(N, kill)          Unsplit a leaf node; remove all children.
int   octGeneration(N)          Level in the tree of node N.
char* octData(N)                Access client data at node N.
char* octSwapData(N, data)      Swap client data at node N.
void  octInitChildIterator(N)   Initialize child iterator for node N.
bool  octNextChild(N1, N2)      Return next child N2 of parent N1.
void  octPrint(N, prnt, strm)   Print the subtree rooted at N.
```

Utility Operations:

```
Str   octDirToStr(Dir)          Return string representation of direction.
Dir   octStrToDir(Str)          Return direction representation of string.
```

Figure 2: The operations exported by the octree ADT. N is a pointer to an octNode; O is a vertex direction; kill is a pointer to a function that removes client data from a node; data is a pointer to a client data structure; prnt is a pointer to a function that prints a client data structure; strm is a pointer to an open FILE structure; Str is a string; and Dir is an octDirection. Also see the appendix.

of a cube's vertices are combinations of the three faces that are incident on each vertex: LDB, LDF, LUB, LUF, RDB, RDF, RUB, RUF. Finally, there is the undefined direction, OMEGA, which appears in Samet [1990A] as the symbol "Ω".

In C, each element of an enumerated type goes into the global namespace. Note that the octDirection type does not follow the convention given above of prepending all such symbols with 'oct'. I deemed that this was too cumbersome, and a future binding of the ADT is planned for C++, where elements of an enumerated type do not pollute the global namespace.

An *octNode* (Figure 1) is a structure that represents each octree node. Although defined in "octree.h", this structure is *opaque*: the client program is not supposed to reference or manipulate the fields (much as the client of the "stdio.h" library is not supposed to access the fields of the FILE structure). Instead, the client uses the provided operations to manipulate *octNode* variables. The provided operations are shown in Figure 2, and in greater detail in the Appendix. The *primary operations* allow standard manipulations of octree nodes. The *utility operations*, *octDirToStr* and *octStrToDir*, are provided to facilitate reading in and printing out octDirections. When possible,

```
octDirection* Vector3ToDirection(Vector3 dir)
/* ensures: Vector3ToDirection(dir) points to a list of quantized
            octDirections that lie near 'dir'.  This list is capped with
            an OMEGA value; client code should loop until the OMEGA
            value is encountered. */
{
   int x_pos, y_pos, z_pos;
   static octDirection quantized_dir_table[8][8] = {
      /* X  Y  Z     Quantized Directions */
      /* R  U  F */  RUF, R, U, F, RU, RF, UF, OMEGA,
      /* R  U  B */  RUB, R, U, B, RU, RB, UB, OMEGA,
      /* R  D  F */  RDF, R, D, F, RD, RF, DF, OMEGA,
      /* R  D  B */  RDB, R, D, B, RD, RB, DB, OMEGA,
      /* L  U  F */  LUF, L, U, F, LU, LF, UF, OMEGA,
      /* L  U  B */  LUB, L, U, B, LU, LB, UB, OMEGA,
      /* L  D  F */  LDF, L, D, F, LD, LF, DF, OMEGA,
      /* L  D  B */  LDB, L, D, B, LD, LB, DB, OMEGA,
   };
   static int access_dir_table[2][2][2] = {0, 1, 2, 3, 4, 5, 6, 7};

   /* Quantize 'dir': R=0, L=1;  U=0, D=1;  F=0, B=1 */
   x_pos = (dir.x > 0) ? 0 : 1;  /* ? R : L */
   y_pos = (dir.y > 0) ? 0 : 1;  /* ? U : D */
   z_pos = (dir.z < 0) ? 0 : 1;  /* ? F : B */

   return(quantized_dir_table[access_dir_table[x_pos][y_pos][z_pos]]);
}
```

Figure 3: Operation to convert from a Vector3 direction into a list of quantized octDirections.

these operations are implemented in-line, which means the client does not lose efficiency from using the provided operations instead of accessing the octNode fields directly.

Quantizing a Direction. Note that any client of the octree ADT must provide a method for mapping the client implementation of a direction into an octDirection. This method is not provided by the octree ADT because different clients might have different ways of implementing a direction. In my implementation, the whole collision detection system is the client of the octree ADT.

In Hook, a direction is implemented as a *Vector3* — a structure consisting of the three floating point elements *x*, *y*, and *z*. Figure 3 shows *Vector3ToDirection*, the operation that converts between a Vector3 direction and an octDirection. Essentially, the routine splits 3-space into two half-spaces along each axis. If the x-component is greater than 0, the direction is R(ight), otherwise it is (L)eft. Likewise, if the y-component is greater than 0, the direction is U(p), otherwise it is (D)own. And if the z-component is greater than 0, the direction is B(ack), otherwise it is (F)ront. Based on these computations for each major axis, Vector3ToDirection returns a list of the octDirections that lie approximately within the half-space containing the Vector3 direction. Note that there is nothing inherently correct about these particular assignments; they simply happen to match Hook's rendered images. Other mappings would also work.

Neighbor-Finding Operations. Samet [1990A] gives elegant and efficient methods for finding the neighbors of an octree node. This is a non-trivial problem because the structure of an octree does not provide neighbor information: spatially adjacent nodes of an octree might be far apart in the tree's hierarchy. In the octree ADT, the octDirection data type forms the basis of the neighbor-finding operations. Figure 4 gives the neighbor-finding operations exported in the file "neighbor.h" (also see

Low-Level Neighbor-Finding Operations

```
bool octAdj(I,O)                True iff O is adjacent to direction I.
O    octReflect(I,O)            Return vertex reflected in direction I from O.
face octCommonFace(I,O)         Return face common to O in direction I.
edge octCommonEdge(I,O)         Return edge common to O in direction I.
```

Neighbor-Finding Operations

```
N    octGtEqFaceNeighbor(P,I)   Return P's >= face neighbor in direction I.
N    octGtEqEdgeNeighbor(P,I)   Return P's >= edge neighbor in direction I.
N    octGtEqVertexNeighbor(P,I) Return P's >= vertex neighbor in direction I.
```

Figure 4: The neighbor-finding operations exported with the octree ADT. I is an octDirection; O is a vertex direction; bool is a boolean value; face is a face direction; edge is an edge direction; N, P are pointers to octNodes. Also see the appendix.

the Appendix). For a given node P, the neighbor-finding operations return a face, edge, or vertex neighbor in direction I from P. In general, each of the neighbor-finding operations ascend the octree until finding the common ancestor of node P and its neighbor N, and then descend the octree until finding the neighbor. The neighbor-finding operations are built on top of four more primitive operations: *Adj*, *Reflect*, *CommonFace*, and *CommonEdge*, which return various types of information about the neighborhood of node O in direction I [Samet 1990A, page 87]. Although the collision detection system does not call these primitive operations directly, they are exported in case the client needs to make use of them.

The implementation of these primitive operations are why Samet's neighbor-finding techniques are so fast, and why it is necessary to quantize a general direction into 26 directions. Each primitive operation is implemented as a table lookup, indexed by its two arguments [Samet 199A, pages 88–90]. Thus, they operate in constant time, and do not reference the octree's representation.

2.2 Building the N-Objects Octree

Initially, the collision detection system begins with one large octree node, the *root* node, which spans the whole scene. The objects are added one-by-one to the root. When the N+1st object is added, the root splits into sub-nodes and inserts its objects into each sub-node. After this splitting, the root inserts all subsequent objects (N+2, N+3, etc.) into its sub-nodes. Each node in the tree follows this same recursive algorithm. Thus, after all the objects have been added to the root, only the leaf nodes contain objects, and each leaf node contains no more than N objects.

Some method is needed to compute the initial size of the root node. In order to detect collisions between objects, the root node needs to encompass the whole space that will be traversed during the desired object motions. The current system uses this heuristic: the root node is set to 150% of the size of the bounding box of all of the objects in their initial positions. This heuristic is flawed, however: it is easy to construct a scenario where objects travel beyond the root node and then collide. In general it is difficult to determine the root's initial size since, depending on the system's dynamics, collisions may alter the direction and velocity of objects.

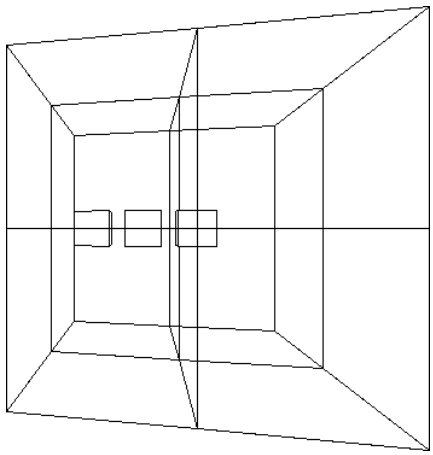

Figure 5: The first frame in a three-frame sequence with N = 2. The outer objects are both traveling towards the central object. Here the right-hand object is about to cross the middle dividing plane. When it does, the left hand nodes must split to follow the constraint of only 2 objects per node.

2.3 Detecting Collisions

At each time step, the system uses the octree to reduce the number of object pairs tested for collision. The system searches the octree depth-first, looking for leaf nodes which contain more than 1 object. For each such node, the node's objects are given to the exhaustive collision detection algorithm. For efficiency, the system prevents multiple instances of two objects being tested in the same time step. It maintains a square table of size NO^2, where NO is the number of objects. For each pair of objects, this table records the last time step when they have been collision tested. For a given pair of objects, if the current time step is equal to the value in the table, the objects have already been tested.

2.4 Splitting and Merging Nodes

As objects move, they enter some octree nodes and leave others. The octree splits and merges nodes as necessary to maintain the N-objects constraint. Figure sequence 5–7 shows an octree splitting and then merging to maintain the constraint N = 2.

At each time step, after all the objects have moved, the system loops through each object, and for each object loops through all the nodes that the object spans. For each node, the system performs two checks:

1) A check for the object leaving the node. When this happens, the system deletes the object from the node, and if the node and its siblings now contain N objects, it merges the siblings together into the parent node.
2) A check for the object entering a neighbor of the node. The system uses the neighbor-finding operations to test if the object has entered any neighbors in the direction of the object's motion. For each such neighbor, the system adds the object to the neighbor, and if the object is the N+1st to be added, splits the neighbor.

The above testing could potentially be expensive, and thus the system uses two methods to reduce the amount of testing required. First, the system tests if the bounding box of an object lies

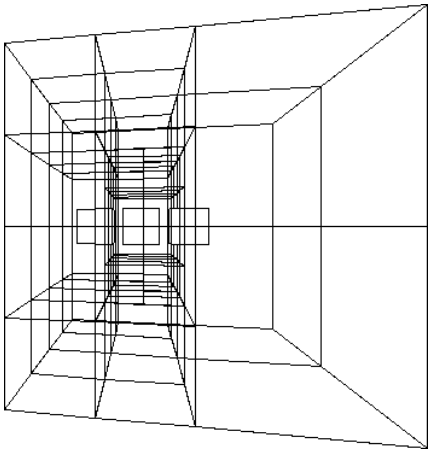

Figure 6: The second frame. Here the right-hand object has crossed the diving plane, resulting in two additional levels of decomposition.

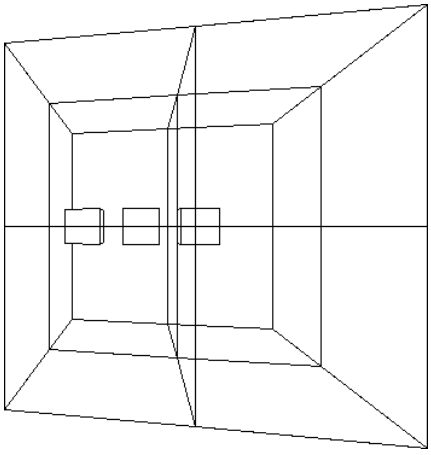

Figure 7: The third frame. Here the right- and left-hand objects have bounced off of the central object, and are now heading outward. Since the right-hand object has now re-crossed the diving plane, the two additional levels of decomposition are no longer needed, and the leaf nodes merge back into their parent nodes.

completely within a node. This is frequently the case, and the test that detects it is very fast. When true, the object lies completely within the node, and neither of the above tests are necessary. Second, the system maintains a counter which is updated for each time step, for each object; this counter is associated with each octree node. The counter quickly tells the system if it has already tested a given neighbor node and a given object. This is necessary for an object which spans multiple nodes, because the neighbor sets of these nodes overlap. It would be inefficient for the system to check for entry into the same neighboring node multiple times.

3 Timing Results

The motivation for the N-objects octree was to reduce the $O(n^2)$ processing time required by the exhaustive collision detection algorithm. I ran a series of timing tests to measure the magnitude of

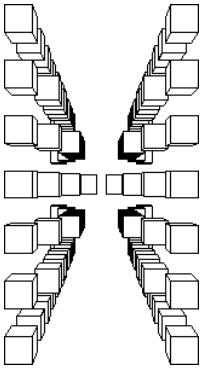

Figure 8: A frame from the test set t3, which contains 98 objects. The planes of objects to the right and the left move towards each other and bounce off.

	time in seconds to perform collision detection										
test set	t0	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
num objects	2	18	50	98	162	242	338	450	578	722	882
N = 5	2.55	12.1	33.9	72.9	120	192	255	344	447	571	726
N = 10	2.85	11.8	34.3	70.8	118	178	246	333	448	559	688
N = 30	2.77	13.6	33.7	70.3	117	181	250	337	443	550	693
N = 50	2.86	12.2	33.9	72.1	116	178	250	337	443	559	674
N = big	2.49	11.9	33.6	69.2	116	183	266	377	474	625	791

Table 1: The time in seconds needed to compute dynamics and collision detection for 200 time steps, for each test set. The first row gives the name of the test set; the second row gives the number of objects in that test set. The remaining rows vary N, the maximum number of objects in each octree node. In the last row the value of N is set to a number much larger than the number of objects in the largest test set. This effectively turns off octree partitioning, and thus gives the running time for the exhaustive collision detection algorithm.

this reduction and to determine, within the Hook system, the algorithm's parameters: above what number of objects does the octree result in substantial time savings? What is a good value for N?

3.1 Method

I developed a script which procedurally generates test sets that are loaded into Hook and used with the collision detection system. A screen shot from one of these sets is shown in Figure 8. The sets consist of two parallel planes of cubes which move towards each other and bounce off in the middle. Each plane consists of a central cube surrounded by concentric rings of cubes; each ring is offset slightly from the middle. The name and number of objects contained in each set are given in the first two rows of Table 1. The script's parameter is the number of rings to generate, which is reflected in the name of each test set. Set *t0* consists of only the two central cubes, which run into each other. Set *t1* consists of the two central cubes, each surrounded by a ring of 8 cubes, resulting in 18 cubes total. Test set *t3*, shown in Figure 8, consists of the central cubes surrounded by three rings, resulting in 98 objects.

Timing data was collected on a Silicon Graphics IRIS Crimson workstation, model number W6-JUR64VGXT, with one 75 MHZ IP17 Processor and 80 Megabytes of main memory. For each test set, the collision detection system was executed for 200 time steps.

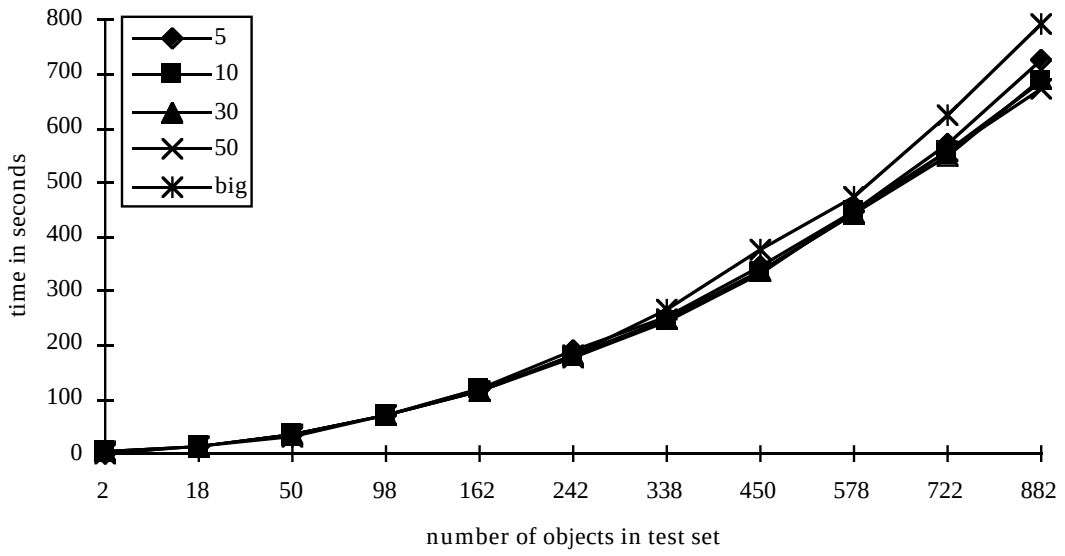

Figure 9: Test set size verses time to perform collision detection. This is a graph of all the data in Table 1.

3.2 Data

The resulting data is the number of seconds required to run the collision detection system on each test set. The data is given in Table 1, and summarized in Figures 9 and 10. Figure 9 graphs all the data; each line represents the collision detection time for various values of N. For the largest value, labeled 'big', N is larger than the number of objects in the largest data set, which is equivalent to using only the exhaustive algorithm. In Figure 9 there is no discernible difference until the 338-object test set. Figure 10 shows only the right-hand side of the data in Figure 9; here the trends are more noticeable. As the test set size increases, the time for the exhaustive algorithm increases faster than the time for the other cases. By the final 882-object test set, the exhaustive case takes the most time, followed by the N = 5 case, followed by the N = 10 and N = 30 cases (which overlap), and closely followed by the N = 50 case.

3.3 Analysis

By the 882-object case it appears that the exhaustive case is the worst, the N = 50 case is the best, and the other cases fall in the middle. However, the trends are not strong enough to suggest statistical significance, and I did not apply significance testing. Hence it is not possible, from this data, to conclude that the octree method is better than the exhaustive method, nor is it possible to determine the optimal value of N for these test sets.

4 Conclusions

The N-objects octree algorithm of Shaffer & Herb [1991] has been implemented using the Hook platform. Although theoretically the octree should run faster than the exhaustive collision detection

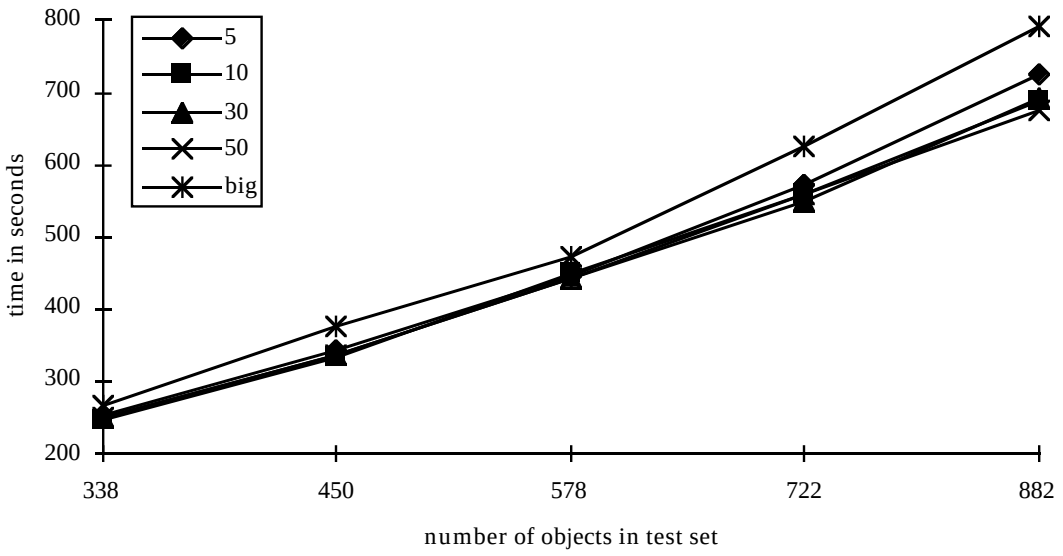

Figure 10: Test set size versus time to perform collision detection. This is a graph of the right-hand side of Figure 9.

system, the timing data collected so far is inadequate to support this. The octree has been implemented as an ADT, which permits the bulk of the current effort to be reused in other projects.

In future work I would like to investigate the small difference in running times between the exhaustive and the octree algorithms. I suspect this occurs because even though the $O(n^2)$ term in the exhaustive algorithm is asymptotically dominant, the system contains very large constant terms, and the current test sets are too small to reveal differences larger than these terms. I suspect that at least part of these terms come from the maintenance of the many linked lists that are associated with the octree. And the internal workings of Hook may add additional terms. I expect that through profiling and study I could reduce these constant terms, and then I would see a greater difference with the current test sets. Or, larger test sets could be used. At the least, this second option requires structuring the system so the testing is script-driven and non-interactive.

I would also like to implement the octree ADT in C++, for use in projects which are already implemented in C++.

References

[Cameron 1985] Stephen Cameron, "A Study of the Clash Detection Problem in Robotics", *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St. Louis, MO, March 1985, pages 488–493.

[Cameron 1990] Stephen Cameron, "Collision Detection by Four-Dimensional Intersection Testing", *IEEE Transactions on Robotics and Automation*, Volume 6, Number 3, June 1990.

[Hahn 1988] James K. Hahn, "Realistic Animation of Rigid Bodies", *SIGGRAPH Annual Conference Proceedings*, Volume 22, Number 4, August 1988.

[Hook 1992] Matthew Lewis, *Hook* [computer program], The Advanced Computing Center for the Arts and Design and the Department of Computer and Information Science, The Ohio State University, 1992.

[Moore & Wilhelms 1988] Matthew Moore and Jane Wilhelms, "Collision Detection and Response for Computer Animation", *SIGGRAPH Annual Conference Proceedings*, Volume 22, Number 4, August 1988.

[Rodgers 1985] David F. Rodgers, *Procedural Elements for Computer Graphics*, McGraw-Hill, 1985.

[Samet 1990A] Hanan Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, 1990.

[Samet 1990B] Hanan Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1990.

[Shaffer & Herb 1991] Clifford A. Shaffer and Gregory M. Herb, "A Real-Time Robot Arm Collision Avoidance System", Virginia Polytechnic Institute and State University, August 15, 1991.

[Shaffer & Herb 1992] Clifford A. Shaffer and Gregory M. Herb, "A Real-Time Robot Arm Collision Avoidance System", *IEEE Transactions on Robotics and Automation*, Volume 8, April 1992, pages 149–160.

Appendix — Octree ADT Operations

This appendix contains a detailed description of the octree ADT operations. For each operation a *prototype* clause, an optional *requires* clause, and an *ensures* clause is given. The *prototype* clause gives the operation's name, and the data types of the operation's return value and arguments. The *requires* clause gives restrictions on the values of the arguments; these restrictions are assumed by the operation, and are necessary for its valid execution. The *ensures* clause states what the operation will achieve, in terms of its arguments, under the assumption that the requires clause is met.

The octree ADT comes in two implementations: a *defensive* implementation, and an *efficient* implementation. The defensive implementation, which is the default, tests that requires clauses are met by means of *assert*(3) statements. As a result of this error-testing, certain operations which could otherwise be implemented in-line are instead implemented as function calls. The *efficient* implementation, which is compiled when the symbol "NDEBUG" is defined, removes all such error checking, and in-lines all operations for which inlining is possible.

Client-Supplied Operations

```
prototype: void ItemDisposer(char* data)
 requires: data access a client data structure.
  ensures: data is disposed; any allocated memory is returned.

prototype: void ItemPrinter(FILE* strm, char* data)
 requires: strm is an open FILE*, data accesses a client data structure.
  ensures: A textual representation of data is printed to strm.
```

Primary Operations

```
prototype: octNode* octCreate(void)
  ensures: Returns a pointer to a new octree root node.
```

```
prototype: void octDispose(octNode* N)
 requires: octParent(N) == NULL.
           The octree rooted at N is empty of client data.
  ensures: The octree rooted at N is disposed.

prototype: void octClear(octNode* N, ItemDisposer* kill)
  ensures: All the client data in the subtree rooted at N is removed, making
           it an empty subtree.

prototype: octNode* octParent(octNode* N)
  ensures: octParent(N) == N's parent, or NULL if N is a root.

prototype: octNode* octChild(octNode *N, octDirection O)
 requires: O is a vertex.
  ensures: octChild(N, O) == N's O child, or NULL if N is a leaf.

prototype: octDirection octChildType(octNode* N)
  ensures: If N is a root node, octChildType(N) == OMEGA. Otherwise,
           octChildType(N) == the child type of N in octParent(N).

prototype: boolean octIsSplit(octNode* N)
  ensures: octIsSplit(N) == TRUE if N is an internal node;
           octIsSplit(N) == FALSE if N is a leaf node.

prototype: void octSplit(octNode* N)
 requires: octIsSplit(N) == FALSE.
  ensures: octIsSplit(N) == TRUE; N is split and given children.

prototype: void octJoin(octNode* N, ItemDisposer* kill)
 requires: octIsSplit(N) == TRUE.
           kill != NULL or the subtree rooted at N is empty of client data.
  ensures: octIsSplit(N) == FALSE; N's children are cleared of any
           client data and removed.

prototype: char* octGeneration(octNode* N)
  ensures: octGeneration(N) == the generation of node N.  The generation
           of the root node is 0, the root's children are 1, their
           children are 2, and so on.

prototype: char* octData(octNode* N)
  ensures: octData(N) == the client data stored at node N.

prototype: char* octSwapData(octNode* N, char* newData)
  ensures: The pointer to client data newData is swapped with N's current
           client data pointer, which is returned.

prototype: void octInitChildIterator(octNode* N1)
 requires: octIsSplit(N1) == TRUE
  ensures: An iterator is initialized that allows every child of N1 to
           be visited.

prototype: boolean octNextChild(octNode* N1, octNode** N2)
 requires: octInitChildIterator(N1) has been called.
  ensures: octNextChild(N1, N2) == TRUE if *N2 is a pointer to a child
           node of N1 which has not been visited since child iterator
           initialization; otherwise all children have been visited and
           FALSE is returned.

prototype: void octPrint(octNode* N, ItemPrinter* prnt, FILE* strm)
 requires: prnt is an ItemPrinter or NULL, strm is an open FILE*.
  ensures: A textual representation of the subtree rooted at N is printed to
           strm using prnt to print the data located at each node.
```

Utility Operations

```
prototype: char* octDirToStr(octDirection Dir)
  ensures: octDirToStr(Dir) is a string representation of Dir.

prototype: octDirection octStrToDir(char* Str)
 requires: Str is a valid upper- or lower-case string representation of a
           direction.
  ensures: octStrToDir(Str) is the octDirection representation of Str.
```

Low-Level Neighbor-Finding Operations

```
prototype: boolean octAdj(octDirection I, octDirection O)
 requires: O is a vertex.
  ensures: octAdj(I,O) == TRUE iff O is adjacent to the Ith face, edge,
           or vertex of O's containing block.

prototype: octDirection octReflect(octDirection I, octDirection O)
 requires: O is a vertex.
  ensures: octReflect(I,O) yields the CHILDTYPE value of the block of
           equal size (not necessarily a sibling) that shares the Ith
           face, edge, or vertex of a block having CHILDTYPE value O.

prototype: octDirection octCommonFace(octDirection I, octDirection O)
 requires: I is an edge or a vertex, O is a vertex.
  ensures: octCommonFace(I,O) yields the type of the face of O's
           containing block, that is common to octant O and its neighbor
           in the Ith direction.

prototype: octDirection octCommonEdge(octDirection I, octDirection O)
 requires: I is a vertex, O is a vertex.
  ensures: octCommonEdge(I,O) yields the type of the edge of O's
           containing block, that is common to octant O and its neighbor
           in the Ith direction.
```

Neighbor-Finding Operations

```
prototype: octNode* octGtEqFaceNeighbor(octNode* P, octDirection I)
 requires: I is a face.
  ensures: If such a node exists, octGtEqFaceNeighbor(P,I) == the
           face-neighbor of node P, of size greater than or equal to P,
           in direction I.  Otherwise, octGtEqFaceNeighbor(P,I) ==
           NULL.

prototype: octNode* octGtEqEdgeNeighbor(octNode* P, octDirection I)
 requires: I is an edge.
  ensures: If such a node exists, octGtEqEdgeNeighbor(P,I) == the
           edge-neighbor of node P, of size greater than or equal to P,
           in direction I.  Otherwise, octGtEqEdgeNeighbor(P,I) ==
           NULL.

prototype: octNode* octGtEqVertexNeighbor(octNode* P, octDirection I)
 requires: I is a vertex.
  ensures: If such a node exists, octGtEqVertexNeighbor(P,I) == the
           vertex-neighbor of node P, of size greater than or equal to
           P, in direction I.  Otherwise, octGtEqVertexNeighbor(P,I) ==
           NULL.
```