Efficient Large-Scale Sweep and Prune Methods with AABB Insertion and Removal

Daniel J. Tracy*
Experimental Game Lab
University of California San Diego

Samuel R. Buss†
Department of Mathematics
University of California San Diego

Bryan M. Woods‡
Department of Cognitive Science
University of California San Diego

ABSTRACT

We introduce new features for the broad phase algorithm *sweep and prune* that increase scalability for large virtual reality environments and allow for efficient AABB insertion and removal to support dynamic object creation and destruction. We introduce a novel segmented interval list structure that allows AABB insertion and removal without requiring a full sort of the axes. This algorithm is well-suited to large environments in which many objects are not moving at once. We analyze and test implementations of sweep and prune that include subdivision, batch insertion and removal, and segmented interval lists. Our tests show these techniques provide higher performance than previous sweep and prune methods, and perform better than octrees in temporally coherent environments.

Index Terms: I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality;

1 INTRODUCTION

Collision detection has broad applications including virtual reality systems, computer games, robot motion planning, physical modeling, and training simulators. Modern systems are capable of detecting physical interference between thousands of objects, each of which is composed of thousands of polygons, in real time. Many collision detection subsystems are composed of two primary stages, called the 'broad phase' and 'narrow phase'. The broad phase takes as input all objects in the environment and produces a list of pairs of objects that are sufficiently near that they could possibly intersect. The narrow phase takes this list as input and processes each object pair to determine collision status, identifying portions of the geometry which interfere. The present paper discusses improved broad phase algorithms based on extensions to sweep and prune and on spatial subdivision. These new algorithms are particularly useful for environments with many unmoving objects and for environments where objects can be created and destroyed.

Bounding volumes are typically used in the broad phase, usually spheres or axis-aligned bounding boxes (AABBs). Many systems utilize spatial subdivision for the broad phase [3, 18, 12, 28, 15, 2, 27, 19, 11]. Spatial subdivision algorithms only check for intersection among shapes occupying the same spatial regions. *Sweep and prune* is an alternative technique that sorts the projected extents of AABBs onto each cartesian axis and incrementally sorts them [1] to determine AABB overlap. Non-incremental sorting methods have also been used in the broad phase [16, 14, 22].

The incremental sweep and prune technique, hereafter referred to as sweep and prune, was first published by Baraff [1]. An implementation by Cohen et al. as part of iCollide [6] used a *local sort* mechanism, useful for environments with few moving objects. Coming and Staadt [7] adapted sweep and prune to an *event-driven* broad phase approach for environments whose objects follow simple known paths by pre-computing and scheduling swap events. They also have work with sweep and prune that eliminates temporal aliasing to support continuous collision detection [8]. More details on sweep and prune extensions are provided in Section 2.

Some systems employ pre-computed hierarchies of bounding volumes over static geometry [13, 23, 29]. This paper differs from these works in that we focus upon "object-level" pruning, where it is assumed that each object is moving independently and a stable hierarchy cannot be formed.

This paper reviews several augmentations to sweep and prune presently in use but not represented in the literature. First, we discuss the utility of using multiple sweep and prune instances in a subdivided environment to reduce swapping behavior, accelerate AABB insertion and removal, and allow parallelization. Excess swapping behavior directly impacts the scalability of sweep and prune, and this method can accelerate the algorithm by an order of magnitude or more in large environments. Second, we present batch methods that reduce the overhead of AABB insertion and removal events. Third, we advocate an event-based output interface that notifies clients of *changes* in collision status, and discuss applications for which this increases efficiency.

In addition, we present a novel technique to accelerate AABB insertion and removal using *segmented interval lists*. This method imposes a hierarchy over the interval lists for searching and allows insertions and erases of extrema without consulting or modifying the entire list. This technique performs better than batch insertion and removal in large environments with many objects at rest and small numbers of insertions and removals. We discuss implementation and show experimental results for all of these techniques.

2 SWEEP AND PRUNE METHODS

Sweeping plane is a broad phase algorithm that finds overlaps between AABBs by sorting the projected extents of boxes on a cartesian axis. The overlaps between boxes on that axis can be determined by a second pass over this list. During this pass, two data structures are maintained: a set E of objects whose minimum has been encountered but whose maximum has not, and a set O of object pairs which accumulates overlaps. At each node, if we encounter the minimum for some object c, we add c to E, or we remove c from E if a maximum. In addition, at every maximum for an object c, we add to O every object in E paired with c. This processing is usually performed on a single axis, and object pairs in the set O are checked for overlap in other dimensions.

Sweep and prune, developed by Baraff [1] and implemented by Cohen et al. [6], improves upon sweeping plane by exploiting temporal coherence. Because object positions often change little each cycle, the projected sort order on each axis and the output remain similar between cycles. By retaining the sorted interval lists and results, and performing incremental changes for movement, performance is dramatically improved. An *insertion sort* [17] is used,

*dtracy@cs.ucsd.edu, daniel.joseph.tracy@gmail.com
†sbuss@math.ucsd.edu
‡woods.bryan@gmail.com

which has an expected linear running time on nearly sorted sequences. Swaps occur between adjacent elements, during which AABB overlap tests are triggered.

There are three primary data structures in a sweep and prune implementation. A sorted list of intervals for each axis contains one minimum and one maximum for each object. In addition, a node exists for each object which contains references to the object's minima and maxima in the sorted lists, six references in all, called the 'AABB list'. When an object must update its AABB due to movement, it updates the values through these references. Finally, the 'candidate set' is the set of object pairs that overlap in all three dimensions. An example of the data structures is shown in Figure 1.

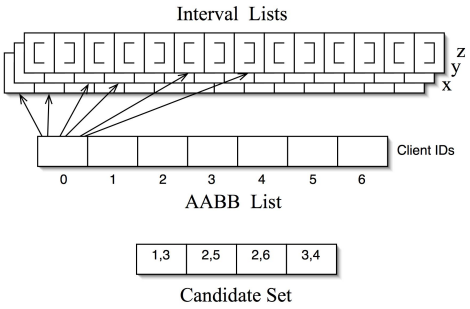

Figure 1: Sweep and Prune Data Structures

The operation of sweep and prune proceeds as follows. Between each processing cycle, objects undergoing motion update their AABB extents, writing them into the interval lists, which will now be unsorted. To resort each list, an insertion sort is performed. During this sort pass, swap events may occur between adjacent objects, causing their AABBs to be compared for full three-dimensional overlap. Swaps that change overlap status cause the candidate set to be updated by adding or removing object pairs. In this way, the amount of work performed for a coherent environment is simply a pass over the interval lists plus some amount of work for each swap.

The sweep and prune method outlined above is as originally detailed by Baraff [1]. The iCollide [6] implementation uses a triangular bit matrix to keep track of overlap status for each dimension rather than perform AABB comparisons on each swap. This requires $O(n^2)$ space but reduces the work required during swaps. More significantly, iCollide optionally uses a *local sort* mechanism, which sorts each extrema as they are updated. This eliminates the need to sort the lists from beginning to end (a *global sort*), and improves performance when few objects move.

Coming and Staadt describe a kinetic sweep and prune [7]. For environments in which object paths follow a closed-form function, swap times are pre-computed for each adjacent pair, eliminating the need to sample positions each simulation cycle. Coming and Staadt also use sweep and prune for continuous collision detection by eliminating temporal aliasing [8], while addressing the high-velocity problem with extended bounding boxes. The present work differs from these papers in that we do not assume any knowledge of object paths and our algorithm is insensitive to changes in velocity; however, it would be possible to use extended bounding boxes to eliminate temporal aliasing with the algorithms presented in Section 3.

2.1 Analysis of Sweep and Prune

Baraff [1] indicated that sweep and prune has complexity $O(n+s)$, where n is the number of objects and s is the number of swaps performed. Considering swaps that do not produce a change in the candidate set to be negligible, he concluded that running time was effectively $O(n+o)$, where o is the number of overlapping pairs.

This time bound does not reflect the cost of swap events in a large simulation, however. It also does not allow for insertion and removal of AABBs during simulation, and the overhead of maintaining the candidate set is not considered. We analyze local-sort based sweep and prune in k-dimensional space taking these factors into account. More importantly, we show that the s term increases super-linearly with object count.

Symbol	Meaning
n	total AABBs
k	dimensions (typically 2 or 3)
m	migrations (insertions + removals)
u	updates (moving AABBs)
s	swaps
o	AABB pairs overlapping
e	changes in o

Table 1: Definition of terms for analysis

The total complexity of sweep and prune is

$$O(uk+sk+e+mnk+o),$$

where u represents position updates, e is the number of AABB pairs that transitioned to or from an overlapping state on this processing cycle, o is the total number of overlapping object pairs, and m is the sum of inserted and removed AABBs. The uk term represents movement updates, which write a value for the k dimensions of all u updated objects. The term sk is work for swaps caused by movement: for each swap, k dimensions are compared for overlap. For e object pairs changing collision status, e work is required to maintain a hashing structure. For each of the m insertions and removals of AABBs, $O(n)$ expected swaps occur in each of k dimensions to place them, yielding mnk. AABB tests are only required for swaps in one of these dimensions during insertion and removal. The o term represents communicating the candidate set to the client.

The expected number of swaps s can be further analyzed based upon the density (extrema per unit axis length) of the interval lists and average velocities. When $k > 1$, sweep and prune increases extraneous swap behavior super-linearly with respect to AABB count. To quantify how the number s of swap events increases with AABB count, we assume that a set of AABBs are uniformly distributed in a k-dimensional hypercube. The volume of the hypercube is increased with the number of AABBs so as to maintain a constant object density. Let E be a k-dimensional hypercube environment with n AABBs and a total volumetric density of V. Let l_E be the length of each side of E. Let the hypercube E' have the same volumetric (object) density V as E, but with fn AABBs, where $f > 1$. The total number of intervals in E is $2n$ on each of the k axes, whereas for E' it is $2fn$. Letting c be the mean volume per AABB, we have

$$V = nc/l_E^k = fnc/l_{E'}^k.$$

Thus, the ratio of $l_{E'}$ to l_E is $\sqrt[k]{f}$.

To summarize, E' has a factor f times as many extrema per axis and only $\sqrt[k]{f}$ times the length for each axis. This means that when the number of objects increases by a factor f, the number s of swaps on the axes can be estimated to increase by a factor

$$f/\sqrt[k]{f} = f^{1-1/k}.$$

Therefore, s can be estimated as $ukn^{1-1/k}$ under the assumptions of constant average object velocity and constant density of uniformly distributed objects. When all objects are moving, $u = n$, so the runtime for the traditional sweep and prune becomes

$$O(k^2 n^{2-1/k} + mnk + e + o).$$

This analysis is supported by the simulations shown in Figure 2. These measurements were taken with cubical objects all moving in a uniform environment and no insertion or removal events. All cubes are the same size, have the same velocity magnitude in random directions, and bounce off the environment's boundaries. The otherwise necessary tracking and reporting features were disabled, as they disturb measurement of swap behavior. Curve-fitting these data points using least squares fitting produces an estimated power of 1.76, close to the $n^{5/3}$ being claimed.

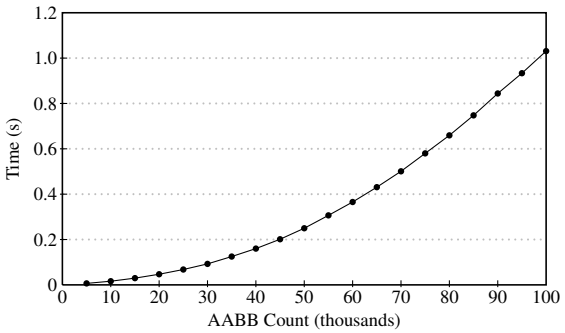

Figure 2: Measured SAP Growth Characteristic

To summarize, we've identified several weaknesses of the traditional sweep and prune algorithm. First, the algorithm does not scale linearly due to increased swap behavior for each extrema when object count increases. Secondly, it does not efficiently handle object insertion and deletion.

3 Enhancements to Sweep and Prune

We present enhancements to sweep and prune that address the performance weaknesses analyzed above. Section 3.1 describes a now common hybrid algorithm with both subdivision and sweep and prune[25]. This reduces swap behavior from $O(uk^2n^{1-1/k})$ to $O(uk^2)$ by limiting axial density. In traditional sweep and prune, the extraneous swap behavior is *the majority of the running time* of the algorithm in large environments. This method directly reduces the number of swaps to linear in the number of updates by preventing distant objects from polluting interval lists. It also accelerates AABB insertion and removal by reducing swaps to cell populations and allows the algorithm to be parallelized easily.

Section 3.2 discusses two methods for improving AABB insertion and removal performance. The first method, batch insertion and removal, is well suited to environments in which most objects are moving or large numbers of insertion and removal events will occur every simulation cycle. The second method, segmented interval lists, is novel to this paper and yields better performance when fewer objects are moving and small numbers of insertions and removals occur due to object creation and deletion or moving objects migrating between cells. This method is well-suited to the subdivision plus sweep and prune scheme, which continuously causes small numbers of migrations.

In addition, an event-based output interface is described in Section 3.3 that allows the client to process only relevant subsets of overlaps, which can reduce the $O(o)$ term of complexity from sweep and prune to $O(e)$.

3.1 Subdivision

A hybrid method using both spatial subdivision and sweep and prune can give large performance improvements. The hybrid method uses a 'superstructure' based on a spatial subdivision method, typically a simple grid, in which each cell is actually an instance of sweep and prune operating over a portion of space. Bounding volumes which overlap more than one cell are duplicated into those cells, so care must be taken to eliminate duplicates from output caused by object pairs overlapping in multiple cells. The superstructure retains a mapping from AABBs to the set of cells they reside within for relaying updates and performing erases, and handles migrations between cells caused by movement. Each object has associated with it an integer key for each cell it resides in, as well as a key tracked by the superstructure, which correlates them with client data.

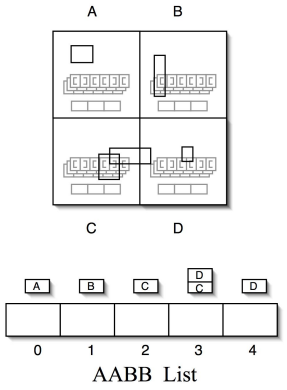

Figure 3: Superstructure of subdivision plus sweep and prune

As compared to a single instance of sweep and prune that covers a large environment, the interval lists of each cell are consequently "thinned" of extrema from distant objects. See Figure 4 for a visual of this effect. If we reduce the object count per cell to a constant, we reduce the expected number of swaps from $ukn^{2/3}$ to uk, as interval density doesn't increase with object count.

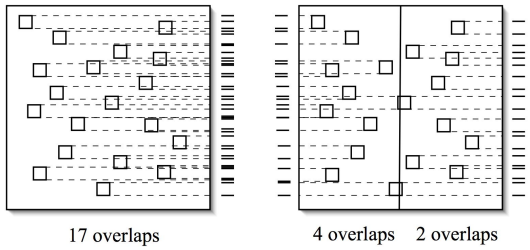

Figure 4: Thinning effect of subdivision

The performance of AABB insertion and removal is also improved by subdivision, reduced from $O(kmn)$ to $O(km)$ if the number of objects per cell can be reduced to a constant. Insertion proceeds through two phases: the superstructure performs cell placement, and then the bounding volume information is swapped into sorted order within each cell. Erases must similarly be swapped out of interval lists. If the number of objects in each cell is bounded, the swapping behavior is bounded by this constant rather than including the whole environment.

This hybrid method lends itself well to being parallelized. In support of thread-parallel execution, movement updates and insertion and removal events buffer all call data within cells. When collisions are queried, the cells are assigned to one of t threads, and each performs sorting in parallel.

Ponamgi et al. [21] developed a hybrid subdivision and sweep and prune algorithm for the narrow phase. That work differs from our discussion in that they intersect cells of two hierarchies and maintain interval lists for each pair of intersecting cells at multiple levels of the hierarchy in order to reduce primitive tests between two models, while we discuss a single structure where sweep and

prune is performed only at the leaves, and is a broad phase algorithm motivated primarily by reducing swaps from distant objects.

In our experiments, the best performance is achieved with large cells and an incremental sort. We achieve optimum performance with about three hundred objects in each cell in a uniformly distributed environment. Further, although adaptive subdivision structures can be used with sweep and prune leaf nodes, a simple grid is typically favored for low overhead in checking cell placement on object position updates. This means that non-uniform object distributions could result in some cells with many objects. This does not pose a serious problem for sorting, but AABB insertion and removal overhead grows linearly with AABB population. This problem is exacerbated by the tendency for more AABB insertion and removal events to occur in largely populated cells, both due to migrations between cells and object creation and deletion. It is clear that additional effort should be made to accelerate object insertion and removal in each cell.

3.2 Insertion and Removal

The primary obstacle to efficient insertion of a new object into the middle of an interval list is that the object may intersect with arbitrarily large AABBs whose extrema can be anywhere within the list. In addition, insertion into or removal from an array-based list requires moving in memory a number of elements proportional to the size of the list. Finally, the placement of extrema must be found efficiently. We discuss two separate techniques to accelerate insertion and removal. The first, batch insertion and removal, has been used in practice but not discussed in the literature[10]. It is efficient when there are many insertions and removals to process and when most objects are moving, as it requires traversal of the interval arrays. The second, which is presented for the first time in this paper, is segmented interval lists. Our technique efficiently processes a small number of insertions and removals when few objects are moving, and limits the number of swaps during insertion or removal regardless of the interval array size. This method is better suited to large virtual environments with many objects at rest.

3.2.1 Batch Insertion and Removal

Instead of individually swapping each inserted object into place, *batch insertion* techniques consolidate this work by presorting insertion extrema to the order they will be processed in the interval array so that they can be integrated in a single pass. Extrema to be removed can also be eliminated from the interval array in batch. This is an efficient solution when a large percentage of objects are moving or when the number of insertions and removals is large.

The implementation we describe here uses a single pass (apart from swaps) over the interval arrays to perform the global sort and integrate all insertion and removal events simultaneously into the same buffer. This optimizes use of the memory hierarchy and is more ideal than multi-pass methods when parallelized due to limited shared memory bandwidth.

Each insertion event generates $2k$ extrema records to insert into k interval lists with coordinate data provided by the client. Rather than performing the insertion immediately, these records are buffered separately for each dimension into lists $I_{i\in[1,k]}$, a minimum and a maximum in each buffer per insertion. Removals are buffered similarly in lists $R_{i\in[1,k]}$. When collisions are reported, for each dimension i, buffers I_i and R_i are sorted. A single pass is then performed over I_i, R_i, and the retained interval list L_i to integrate the population changes, perform the insertion sort, and produce changes in the candidate set. Records to remove are encountered in the interval list L_i in the same order as in R_i. Insertions are processed in the order that their extrema are encountered in I_i. The $O(mnk)$ term of sweep and prune's complexity is reduced to $O(k(m\ log\ m+n))$. A single-space implementation is used to increase cache performance, with the new interval lists replacing the old incrementally within the same array. This is accomplished by "floating" the array of intervals within a slightly larger allocation whose size is bounded by the number of elements plus twice the number of additions on some previous cycle, and performing the algorithm's pass alternatively backward or forward depending upon conditions (See Figures 5 and 6). If a sufficiently large gap exists to accommodate all insertions, a forward pass is performed and the gap is closed. Otherwise, the array is grown by enough elements to accommodate all insertions and a backward pass increases the gap size.

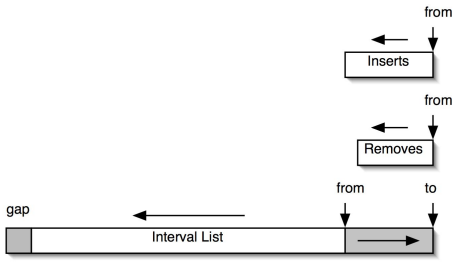

Figure 5: Sorting Backwards

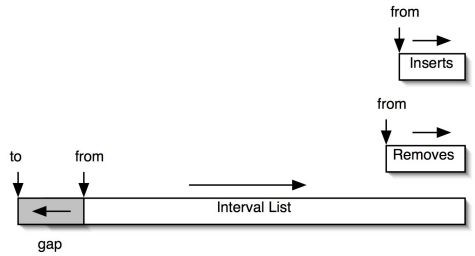

Figure 6: Sorting Forwards

This process produces a valid interval list, but more is required to maintain the candidate set. Inserted objects must be tested for collisions with both retained objects and other inserted objects, and objects being removed must have collisions involving them removed. In order to accomplish this, the inner loop has elements of the non-incremental sweeping plane algorithm. In the inner loop, we maintain three sets to track the identiies of objects intersecting the sweeping lines of the interval list, the insertions list, and the removals list, named S_L, S_I, and S_R, respectively. When an object's first entry is fetched (a minima if going forward), the object's identifier is added to the appropriate set, and it is removed from the set when the second entry is fetched. The set S_L does not include elements being deleted.

When a removed extrema is passed or an insertion is placed into the interval list, the sets provide the context necessary to limit overlap checking to AABBs overlapping this AABB in the dimension being processed. Since objects must overlap in every dimension to affect the candidate set, this processing can be limited to a single axis, with overlap checks occurring for the other axes. The method we use detects all overlap configurations and produces no duplicates in the output. When an object's second entry is copied into the interval list from the insertions list, checks for overlaps between that object and all objects in S_L and S_I are performed, with results added to the candidate set. When the second entry of an item to be removed is passed in the interval list, it is checked for overlap with all objects in S_L and S_R, with results *removed* from the candidate set. In addition, when the second entry of an object is sorted from the interval list into the same list (i.e., due to movement), it is checked against objects in S_I to add to the candidate set and objects

in S_R to remove from it. Pseudo-code for the inner loop is given in Table 2.

```
subfunction process_sets ( set Maintain, set CollideWith, ...
... extrema Point, set_of_pairs Events )
 if Point is minima
  add Point to Maintain
 else
  remove Point from Maintain
  add to Events all {Point, items in CollideWith}

while( not done )
 if L[from_L] = R[from_R]
  if first axis
   process_sets ( S_R, S_L ∪ S_R, L[from_L], uncollideEvents )
  increment from_L & from_R
 else if I[from_I] < L[from_L]
  write I[from_I] to L[to_L]
  if last axis
   process_sets ( S_I, S_L ∪ S_I, I[from_I], collideEvents )
  increment from_I & to_L
 else
  if first axis
   process_sets ( S_L, S_R , L[from_L], uncollideEvents )
  else if last axis
   process_sets ( S_L, S_I, L[from_L], collideEvents )
  insertion sort L[from_L] into L[to_L]
  increment from_L & to_L
end
```

Table 2: Batch Insertion and Removal: Forward Pass

The critical difference between the performance of batch insertion and removal and that of sweeping plane is that a general sort algorithm is not performed on retained data, but only on records of insertion and removal events. This yields performance similar to sweep and prune when few insertion and removal events occur, and smoothly degenerates to sweeping plane performance when large numbers of events occur.

3.2.2 Segmented Interval Lists

We now discuss a new approach named *segmented interval lists*. Traditional sweep and prune uses either linked lists or arrays to store sorted extrema. Segmented interval lists instead use a linked list of small arrays. Another array of pointers allows direct access to each small array and can be binary searched. Since the small arrays may be only partially filled, objects can be inserted and removed without requiring swaps along the entire length of the axis. Segmented interval lists allow small numbers of insertions and removals to occur efficiently by limiting the number of swaps for each event to a constant.

The segmented interval list method works well with a local-sort sweep and prune for large virtual environments in which many objects are at rest, and efficiently handles the smaller number of inter-cell migration events caused by objects moving in a subdivided environment. This method has lower asymptotic complexity than batch insertion and removal when few objects are moving, and the same complexity when all objects are moving ($O(k(m\ log\ n))$ for segmented interval lists vs $O(k(m\ log\ m+n))$ for batch insertion and removal).

In place of lists or arrays, we utilize a hybrid data structure which acts as an *unrolled linked list* [24] with respect to traversal, and a hierarchical structure with respect to insertion. More importantly, it has features that eliminate the need to scan the list beyond the current array segment for overlapping extrema. Refer to Figure 7.

The structure contains a linked list of chunks, which are also

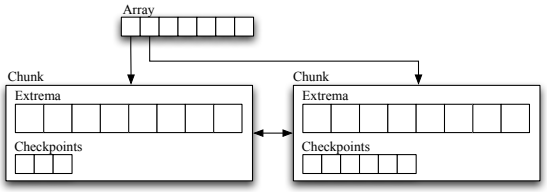

Figure 7: Structure of Segmented Interval List

referred to by an array of pointers. Each chunk has space for a constant number of extrema, but chunks are not necessarily fully populated. The 'checkpoints' set contains the set of object id's which overlap with the 'trailing edge' of the chunk (AABBs whose minima is within this chunk or one to the left, but whose maxima is in a chunk to the right). This yields local information regarding extrema that are arbitrarily far away in the list. This information is only maintained for a single dimension.

Insertion of an extrema involves a binary search on the array to find the earliest chunk with a greater extrema. The new extrema is placed in the last position and insertion-sorted into position within the chunk. After all $2k$ extrema are inserted, the candidate set is maintained by comparing for full-dimension overlap against the set of objects that overlap in the first dimension. This set includes objects with extrema within the same chunk or within the checkpoint set of the chunk occupied by one of the extrema. If the minima and maxima of the inserted object are placed in separate chunks in this dimension, objects in all the chunks between them are compared as well.

Erases involve first performing these full-dimension overlap tests to determine the set of overlaps that exist involving the erased object for removal from the candidate set. The object's extrema are then swapped out of the chunk. This limits the number of swaps for each event to the size of a chunk.

The AABB list contains not only pointers for each of $2k$ extrema, but also a pointer to a chunk for each such extrema, necessary for maintaining them during local sort. During the insertion sort caused by movement updates, each time an extrema swaps the pointer must be updated in the AABB list. In addition, swaps that occur across chunk boundaries cause both the extrema pointer and associated chunk pointer to be updated in the AABB list.

Some work is required to maintain the checkpoints sets. Swaps that occur across chunk boundaries during sort, insertions across multiple chunks, and removal of extrema pairs in separate chunks all invoke changes to the checkpoints. This work is only done for one dimension, and the checkpoints sets in other dimensions can always remain empty.

During sort, a minima swapping leftward into a chunk causes that object id to be added to the checkpoints set, as does a maxima swapping to the right out of a chunk. A minima swapping to the right out of a chunk or a maxima swapping to the left into a chunk causes the object id to be removed from the checkpoints set.

A special case occurs to handle fast-moving objects that may cause their minima and maxima to swap out of relative place temporarily (maxima then minima) and cross a chunk boundary. This would normally cause us to first attempt to remove the object id from the checkpoints set when it is not present and then later add it when the opposing extrema crosses the boundary. To compensate for this problem, when we attempt to remove an object id that is not present in the checkpoints set, we add it to a set R. Whenever an object id is to be added to the checkpoints set, we first check if it is in R and if so, remove it from R instead of adding it to the checkpoints set.

When an object is inserted and its minima and maxima do not map to the same chunk, every chunk from the minima inclusive to

the maxima exclusive adds the object to its checkpoints set. Maintaining a checkpoints set during removals is similar to that for insertions, save that a remove from the set is triggered.

When a chunk must be split due to insertion into a full chunk, a new chunk is allocated and placed next in the list. Approximately half the elements from the end of the full chunk are copied into the new chunk, and the new chunk inherits the old chunk's checkpoints set. The checkpoints set for the old chunk is computed by starting with the original set and modifying this set while traversing the new chunk. Beginning with the end and traversing backwards, when a maxima is encountered, its object id is added to the checkpoints. When a minima is encountered, its object id is removed.

When extrema are erased from a chunk, there is opportunity to potentially merge two neighboring chunks to reduce fragmentation. If the chunk being erased from can be merged with a neighbor and result in population below a threshold, it does so. The resulting chunk inherits the checkpoints set of the chunk further right in the list.

Our tests show that a chunk size of approximately 2^5 gives optimal performance for insertion and removal, while sorting performance benefits somewhat from slightly larger chunk sizes. This is expected to be architecture and implementation dependent, however. There are some simple methods for keeping the chunks unfragmented, primarily by choice of chunk size and insertion and erase policies. Specifically, when an object is erased, it could be swapped backward and across to the previous chunk first, eliminating an element from the previous chunk instead of its own. This decision could be based upon the population of the chunks and the proximity of the element to each side.

Previous systems related to segmented interval lists have used stabbing counters [26] or markers [9] to accelerate AABB insertion and removal. Stabbing counters attempt to limit the number of AABBs that need to be checked for overlap during insertion, but perform poorly if even one large AABB overlaps the insertion point. Markers are similar to our checkpoints, in which the AABB overlap set is tracked at a number of points within each interval list, allowing inserted objects to begin at any marker in a linked-list based implementation. As compared to markers, segmented interval lists provide for insertion allocation in an array-based system, which is expected to perform better on modern hardware. Our system also ensures an upper bound on the number of extrema between checkpoints.

3.3 Event-based Output

Sweep and prune computes the *changes* in overlap status in order to maintain a set of overlapping pairs. Sweep and prune algorithms can therefore provide changes in overlap status efficiently without traversing the set of overlaps, eliminating the o term from its complexity. For cases in which the client must perform some processing each cycle for all overlaps, there is no advantage. However, not all overlaps require continuous processing.

Many applications model a large number of objects, many of which are in a stable 'resting' state where physics calculations are not required until a collision or other disturbance occurs. These environments are ideal for sweep and prune with local sort, as only active objects must be updated and sorted. Because two stable objects need not be checked in the narrow phase each cycle, it would be beneficial if the broad phase provided changes in overlap status, rather than require that all overlaps be produced, traversed, and filtered for relevant candidate collisions. The changes in overlap status are expected to be very small compared to the full number of overlaps in temporally coherent environments.

Another example in which this interface would be beneficial is an advanced physics system such as Timewarp Simulation [20]. In this technique, an optimistic large-step rigid-body simulation is followed by further correction steps using subsets of objects in the simulation based upon collisions. A cycle of updating some objects, detecting collisions, and performing further integration occurs. Using a standard broad phase interface, time is spent each refinement cycle outputting the full set of overlaps to the physics simulation, which must then search for the relevant collisions. An event-based interface provides a small set of events that will only involve objects that have been updated.

A third example occurs when the client requires code to be executed when two AABBs begin or cease to collide, but not continuously during overlap. In this usage, large AABBs can contain many objects without incurring any overhead for maintaining or traversing this set.

The low-bandwidth nature of event-based output is used to accelerate consolidation of overlaps during query. A list of cells receiving updates accumulates cell identifiers until a fraction of the total AABB population is reached. If this limit is reached, all cells are queried, otherwise those listed are queried. The query gathers the *changes* in overlap status from the previous cycle, and a single table maintains the number of cells witnessing the overlap of each object pair. When a witness count changes between zero and one, a collide or uncollide event is provided for the client accordingly. This process has complexity $O(u+e)$.

4 Experimental Results

We have implemented efficient spatial subdivision and sweep and prune methods for performance comparisons. We use an efficient *dynamic octree* [27] as a baseline spatial subdivision technique. Our octree implementation uses dynamic arrays (arrays which may grow by reallocation) to represent object lists, and performs over 80% of its time traversing arrays, indicating near optimality. We also compare sweep and prune implementations that include subdivision using a static grid, the batch insertion and removal technique discussed in Section 3.2.1, and our segmented interval list approach.

Our implementations support any number of dimensions, any comparable data type for spatial position, and any form of client object key for output reporting. This is done through generic programming principles using C++ templates. An infrastructure for testing the correctness and performance of broad phase algorithms has been implemented in C++ and OpenGL. Correctness is tested by optionally comparing all output to a known working algorithm. The source code to our algorithm will be freely available online, along with the testing framework and methods used for comparison.

Both the batch insertion and removal and segmented interval list implementations dynamically determine whether to perform a global sort or local sort in each cell upon every query. Because all updates and events are buffered for parallelization rather than immediately executed, it is apparent upon each query which approach should be utilized. For batch insertion and removal, a global sort occurs if any insertion or removal events are pending or if a large number of updates exist. For segmented interval lists, the total number of insertion, removal, and update events must exceed a threshold for global sort. Although local sort is a significant optimization when few objects are moving, our tests indicate an over 30% speedup using global sort when most objects are moving, justifying this adaptive approach.

All tests are performed on a 2.16 GHz Intel Core 2 Duo machine with 3 GB RAM running Mac OS X 10.5.5. Our synthetic benchmark consists of uniform distributions of AABBs in a cubical environment. Each test uses objects moving randomly with a velocity ten percent of its width per time step in an environment with a volumetric density of five percent. Objects 'bounce' off environment bounds. All results are for a single thread.

4.1 Insertion and Removal Performance

The tests in this section compare the performance of segmented interval lists versus batch insertion and removal. These tests, shown in Figures 8 and 9, do not use subdivision.

The relative performance of batch insertion and removal against segmented interval lists depends upon the extent of activity. Figure 8 shows the total time for sorting movements plus performing one insertion and one removal from the environment each simulation cycle as we vary the number of moving objects among a base of 3000 objects total.

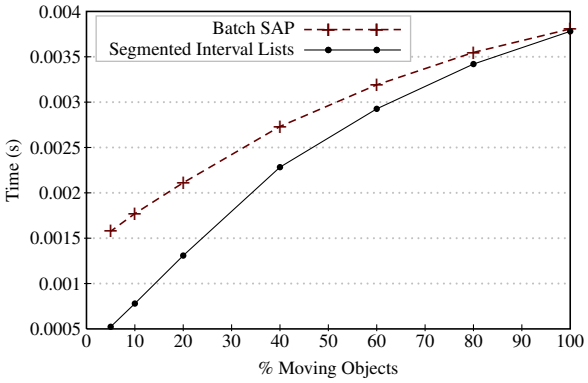

Figure 8: Single Cell, Total time with one insertion & one removal

For a single insertion and removal, segmented interval lists approaches the overhead of batch insertion and removal when most objects are moving and the imposition of a global sort is not a burden. When fewer objects are moving, however, segmented interval lists are much more efficient. With 20% of objects moving, segmented interval lists is 60% faster, while with 5% of objects moving segmented interval lists is three times faster than batch insertion and removal.

In Figure 9, we vary the number of insertions and removals in the context of 150 moving objects among 3000 total (5%). Total time for sorting updates and performing insertions and removals is given.

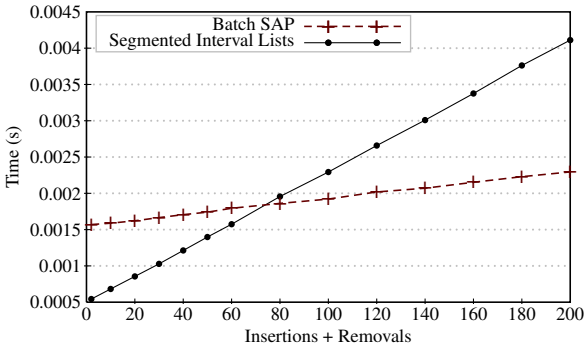

Figure 9: Single Cell, Total time: 3000 objects total, 150 are moving

For small numbers of events, segmented interval lists provide lower overhead, allowing it to process larger environments in real time under coherent conditions. For a single insertion and a single removal, segmented interval lists is almost three times as fast. For ten insertions and ten removals, segmented interval lists is still 90% faster. The cut-off point in this graph is slightly less than 80 events (forty insertions and forty erases).

The vertical translation of the Batch SAP line in Figure 9 is affected by the total objects. It will be lower for less populated environments and higher for more highly populated environments. The shallower slope of the Batch SAP line indicates a lower constant of overhead for processing larger numbers of extrema. The steeper slope of segmented interval lists may be improved by a method that presorts the events and integrates changes into the chunks in order. This consolidates swaps within each chunk when events are high and still reduces swap behavior when events are low. This should produce a performance behavior that smoothly transitions between the best results of both segmented interval lists and batch insertion and removal.

4.2 Aggregate Performance

The remainder of our tests measure the performance of subdivision using a multi-cell grid with sweep and prune cells that use either segmented interval lists or batch insertion and removal. The grid size is optimized based upon object count, ideally about three hundred per cell based upon tests, and is always the same for the two sweep and prune algorithms being compared. The octree's subdivision level is set to have smaller cells for optimum performance.

For the test in Figure 10, ten thousand objects are moving in an environment with one hundred thousand total objects. We vary the number of insertions and removals and measure the total time.

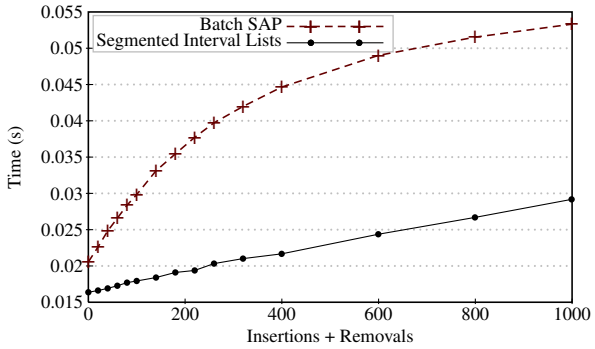

Figure 10: Multi-cell, 100,000 objects, 10,000 Moving

In a large environment supported by the subdivided sweep and prune scheme, batch methods show significant slow-down as insertion and removal events increase at first. This is because the events statistically map to a greater number of cells, each one requiring a global sort. The overhead increase drops off after most of the cells are affected, and the batch method begins to benefit from single-pass integration of multiple events in each cell. Segmented interval list-based cells are much more efficient for lower levels of insertions and removals because they limit the number of swaps for each event. The crossover point for this graph (not shown) is four thousand events (two thousand insertions and two thousand removals). Again, the slope of segmented interval lists could be improved by sorting insert/erase records and consolidating swaps within each chunk.

It should also be noted that the uniform distribution tested is ideal for batch methods, as the segmented technique will perform relatively better as individual cells become overpopulated. The octree is not included because it is insensitive to insertions and removals, although it has generally worse performance than subdivided sweep and prune when few objects are moving.

In Figure 11 we measure performance as we scale an environment and all events proportionately. We perform moving updates on 10% of our total objects, and the insert and remove rate is 0.5% of the total objects each. We test the scale of this environment from ten thousand to two hundred thousand objects.

The segmented interval list system is slightly faster than the batch method at small scales in this test, and becomes almost twice as fast at twenty thousand moving objects among two hundred thousand. The performance advantage widens further if the object distribution becomes non-uniform or the percentage of moving objects is decreased.

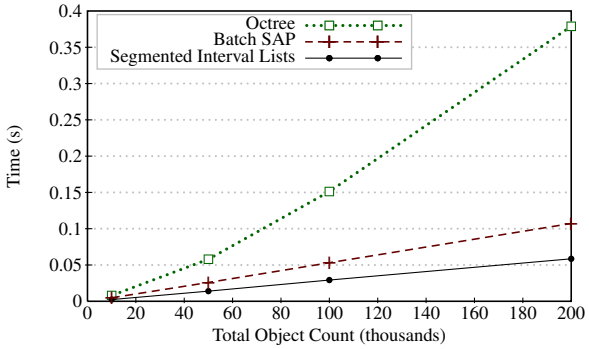

Figure 11: Multi-cell, 10% Moving, 0.5% insertions, 0.5% removals

Although insensitive to insertions and removals, the octree cannot take sufficient advantage of temporal coherence as compared to sweep and prune approaches. Under conditions in which object velocities are low or there are many objects not in motion, sweep and prune methods perform better.

The algorithms presented in this paper are used in the *Scalable City* [4, 5] project, a large virtual reality artwork which is able to test algorithms for large real-time VR systems. Tens of thousands of physical objects are simulated across large landscapes, with a variety of forces, physical effects, and behaviors. A sweep and prune algorithm is used to accelerate many software components by providing proximity information for collision detection and other effects based upon spatial proximity. Most objects in this environment are at rest most of the time, but can become activated immediately based upon interaction. Subdivision, segmented interval lists, and event based output result in much-improved execution times.

4.3 Conclusions

We have presented several enhancements to sweep and prune for broad phase collision detection. Our segmented interval lists, in combination with subdivision, provide sweep and prune with object insertion and removal performance several times faster than batch methods. These techniques perform especially well in large environments with many unmoving objects in which objects can be added to and removed from the system.

Acknowledgements

This work supported in part by NSF grant DMS 0700533.

References

[1] D. Baraff. *Dynamic Simulation of Non-Penetrating Rigid Bodies*. PhD thesis, Cornell University, 1992.

[2] R. Bayer and E. McCreight. *Organization and maintenance of large ordered indexes*, pages 245–262. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[3] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[4] S. Brown. The scalable city project. http://scalablecity.net.

[5] S. Brown. The scalable city. In *SIGGRAPH '07: ACM SIGGRAPH 2007 art gallery*, page 192, New York, NY, USA, 2007. ACM.

[6] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 218, 1995.

[7] D. S. Coming and O. G. Staadt. Kinetic sweep and prune for multi-body continuous motion. *Computers & Graphics*, 30(3):439–449, June 2006.

[8] D. S. Coming and O. G. Staadt. Velocity-aligned discrete oriented polytopes for dynamic collision detection. *IEEE Transactions on Visualization and Computer Graphics*, 14(1):1–12, 2008.

[9] E. Coumans. Physics simulation forum. http://www.bulletphysics.com/Bullet/phpBB3/viewtopic.php?f=4&t=327, May 2006.

[10] E. Coumans. Sap discussion. http://www.bulletphysics.com/Bullet/phpBB3/viewtopic.php?p=&f=&t=209, January 2006.

[11] M. Eitz and G. Lixu. Hierarchical spatial hashing for real-time collision detection. In *SMI '07: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, pages 61–70, Washington, DC, USA, 2007. IEEE Computer Society.

[12] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[13] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[14] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 25–32, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[15] A. Guttman and M. Stonebraker. R-trees: A dynamic index structure for spatial searching. Technical Report UCB/ERL M83/64, EECS Department, University of California, Berkeley, 1983.

[16] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, 1995.

[17] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.

[18] C. Levinthal. Molecular model-building by computer. *Scientific American*, 214(6):42–52, 1966.

[19] R. G. Luque, a. L. D. C. Jo and C. M. D. S. Freitas. Broad-phase collision detection using semi-adjusting bsp-trees. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 179–186, New York, NY, USA, 2005. ACM Press.

[20] B. Mirtich. Timewarp rigid body simulation. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 193–200, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[21] M. K. Ponamgi, D. Manocha, and M. C. Lin. Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics*, 3(1):51–64, 1997.

[22] S. Rabin. Recursive dimensional clustering: A fast algorithm for collision detection. In *Game Programming Gems 4 (Game Programming Gems Series)*. Charles River Media, Inc., Rockland, MA, USA, 2004.

[23] S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies, 2002.

[24] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*, pages 185–191, 1994.

[25] P. Terdiman. Physics simulation forum. http://www.bulletphysics.com/Bullet/phpBB3/viewtopic.php?f=4&t=1329, August 2007.

[26] G. van den Bergen. *Collision Detection in Interactive 3D Environments*. Morgan Kaufman, 2003.

[27] B. C. Vemuri, Y. Cao, and L. Chen. Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum*, 17:121–134, June 1998.

[28] G. Yuval. Finding near neighbors in k-dimensional space. *Inf. Process. Lett.*, 3(4):113–114, 1975.

[29] X. Zhang, S. Redon, M. Lee, and Y. J. Kim. Continuous collision detection for articulated models using taylor models and temporal culling. *SIGRAPH*, 2007.