Comparison of Common Broad Phase Collision Detection Algorithms

Christopher Dang
Sasha Demyanik
Sebastian Sanchez
Ethan Schluze
School of Engineering and Computer Science
Washington State University
Vancouver

***Abstract*—The abstract goes here.**

I. Introduction

The introduction goes here

II. Brute Force

Most applications that require collision detection use the brute force method as the first type of broad phase collision detection implementation. The brute force method serves as the basis to almost all of the broad phase collision detection algorithms. The reasoning is that it solves the most basic of problems, detecting clipping between two objects in a world. If the objects do not ever clip with each other then there is no need for any collision detection. But if one or more objects do clip each other then we can group them as a collection of objects. For each collection used by the brute force algorithm and other collision detection algorithms we iterate the collection's element with one another to detect whether they clip or not.

The difference between the brute force algorithm and the other collision detection algorithms is how the other algorithms manage the size of the collection. Managing the size of the collection is the most important aspect to broad phase collision detection algorithms because when brute force is called the number of pairs of objects that need to be tested grow quadratically with respect to the size of the collection.

To test all pairs of objects in a collection would require n^2 tests for n items in a collection. We can drop that amount by roughly half by only testing unique pairs of objects. This is shown in Algorithm 1.

Algorithm 1 Brute Force - Brute Force

```
1: function BRUTE FORCE(E)
2:     for i = 0 to E.length do
3:         for j = i to E.length do
4:             TEST COLLIDE(E[i],E[j])
5:         end for
6:     end for
7: end function
```

Even with the reduction, both methods result in a running time of $O(n^2)$ for n objects in a collection.

III. Spatial Partitioning Index

Spatial partition index (SPI) (partition is usually omitted and also known as grid) is the simplest to implement and understand of most broad phase collection algorithms. The purpose of these algorithms is to limit the collection size before calling the brute force algorithm. SPI creates a collection of collections. Each subcollection has fixed and defined bounds based on the grid size. The number of subcollections depends on the world size and the grid size. A limitation of SPI is that it is bounded and does not handle a arbitrarily large world size without also having an arbitrarily large grid size. Having both a large world size and grid size may defeat the purpose of keeping the number of elements in a subcollection small.

With a world size given as width and height of the world - the number of x grids: grids that stretch from minimal x bound of the world to the maximum x bound of the world, is given as the width of the world divided by the grid size; the number of y grids: grids that stretch from the minimal y bound of the world to the maximum y bound of the world, is given as the height of the world divided by the grid size. This is shown on lines 2, 3 in Algorithm 2.

Algorithm 2 Initialize - Spatial Partitioning Index

```
1: function INIT(width, height, size)
2:     xGrids ← CEIL(width/size)
3:     yGrids ← CEIL(height/size)
4:     G[][] ← G[x][y]
5: end function
```

Objects that are larger than the grid size or are placed across the bounds of the grids must be placed and referenced into multiple subcollections. This is done with Algorithm 3. We start at the index of the objects top-left bound and incrementally increase the grid position in the x and y directions and place them into the correct container. This partitioning part, also known as the hashing part, hashes the object's reference into a bucket based on the objects spatial positioning. This is usually done by simply dividing the object's spatial position by the grid size.

With a small grid or large world, this matrix of grids can grow extremely large and may become computationally im-

possible to hold in memory. An extremely small grid size can result in unnecessary thrashing (performance wise) caused by trying to fit large objects into many grid slots (subcollections). If we consider a grid size of a unit size of 1, and objects consisting of a height and width a large multiple of the unit size, then for each and every object in the world we must place a reference of the objects into its width times its height number of grid slots. If this is done over hundreds of objects, then most of computational time will be spent in the adding the element's spanning section.

Thus it is important to hand tune the grid size so that the number of spanning calls are limited and the number of elements in the subcollections are kept to a minimum. This is a balancing act between the number of objects in a collection and the number of spanning calls are needed.

Algorithm 3 Spanning Add Element - Spatial Partitioning Index

```
1: function SPANNING ADD ELEMENT(E)
2:     for i = 0 to E.length do
3:         for y ← E[i].MINY to E[i].MAXY + size do
4:             for x ← E[i].MINX to E[i].MAXX + size do
5:                 row ← y/size
6:                 col ← x/size
7:                 if row < yGrids and col < xGrids then
8:                     G[x][y].ADD(E[i])
9:                 end if
10:            end for
11:        end for
12:    end for
13: end function
```

After each object in the world has been added to the collection of collections, we iterate through each subcollection and call the brute force method for each subcollection. This is shown in Algorithm 4.

Algorithm 4 SPI - Spatial Partitioning Index

```
1: function SPI(G[][])
2:     for i = 0 to xGrid do
3:         for j = 0 to yGrid do
4:             BRUTE FORCE(G[i][j])
5:         end for
6:     end for
7: end function
```

IV. Sweep and Prune

This broad phase collision detection algorithm Sweep and Prune is very different from Grid and Quad Tree. Sweep and Prune builds its collection based on the an objects bound itself. Thus there is no spatial bound on the world required when using Sweep and Prune.

The basic premise to Sweep and Prune is a greedy method of organizing each object by their minimum bound on one or more axis. In this case for Algorithm 1 and 2 only 1 axis has been chosen, the x axis.

Algorithm 5 Sweep - Sweep and Prune

```
1: function SWEEP(E)
2:     for i = 0 to E.LENGTH do
3:         PRUNE(F, E[i])
4:         F.ADD(E[i])
5:         BRUTE FORCE(F)
6:     end for
7: end function
```

For each object in the world, we obtain the furthest away object on one axis and continually grab the next object that is the next furthest away. We have an active collection **F** of items and will call the brute force method on that collection. Before adding to the active collective **F**, we check if any object maximum bound is less than the new object's minimal bound. By doing this we ensure that any newly added object has a chance of colliding with another object. As an equality comparison: $e_1.max < e_2.min$, where e_2 is the newly added object. This is done as the prune step. This is shown in line 3 and 4 of Algorithm 6.

Algorithm 6 Prune - Sweep and Prune

```
1: function PRUNE(F, e)
2:     for i = 0 to F.length do
3:         if F[i].MAXX < e.MINX then
4:             F.REMOVE(i)
5:         end if
6:     end for
7: end function
```

After pruning and inserting a new element into the active list, we call the brute force method on the active list. This is done continually until all objects have been iterated and added to the active list. This is shown in Algorithm 5.

V. Quad Tree

The Quad Tree algorithm is very similar to the Grid algorithm. The main difference between the two is data structure behind each algorithm. For Quad Tree like the name implies it is backed by a 4-ary tree, the Grid uses a list or array. The Quad Tree in this implementation has spatial bounds, and all collections are placed into the leaves.

For this collision detection algorithm we must first populate the tree with objects. This is done with the Add Element method in Algorithm 7. We add elements in a node until the node fills up. Once the node fills up it splits up and empties its elements. Doing this ensures that the elements are stored at the leaves of the tree.

When splitting, we create four subnodes whose boundaries are based off the cardinal directions of the current origin. Each subnode is labeled as a quadrant in the euclidean coordinate system, whose axis are flipped based on the ST/UV coordinate system. Each subnode has its own origin based on its own boundaries. Then each object is added to one or more of the

Algorithm 7 Add Element - Quad Tree

```
1: function ADD ELEMENT(e)
2:     if Quadrant ≠ nil then
3:         SPANNING ADD ELEMENT(e)
4:     else
5:         E.ADD(e)
6:         if Count + 1 > CountMax then
7:             SPLIT
8:             Count ← 0
9:         else
10:            Count ← Count + 1
11:        end if
12:    end if
13: end function
```

Algorithm 8 Split - Quad Tree

```
1: function SPLIT
2:     QI ← new QUADTREENODE(xOrigin + xOrigin ∗ 0.5, yOrigin − yOrigin ∗ 0.5)
3:     QII ← new QUADTREENODE(xOrigin − xOrigin ∗ 0.5, yOrigin − yOrigin ∗ 0.5)
4:     QIII ← new QUADTREENODE(xOrigin − xOrigin ∗ 0.5, yOrigin + yOrigin ∗ 0.5)
5:     QIV ← new QUADTREENODE(xOrigin + xOrigin ∗ 0.5, yOrigin + yOrigin ∗ 0.5)
6:     for i = 0 to E.length do
7:         SPANNING ADD ELEMENT(E[i])
8:     end for
9: end function
```

subnodes based on the object's dimensions with the Spanning Add Element method on line 7 of Algorithm 8.

With the assumption that the object does not go out of the bounds of the world, we check an object's maximum bounds in a certain cardinal direction and add it to the corresponding subnode.

For Quad Trees that has a limit of elements in the nodes, the objects that should be placed in the nodes should have a unit size dimension, it is isn't required, but the use of QuadTrees is recommended for objects of that type. This can be a problem when many objects with large dimensions are added to the tree such that they can propagate 4-fold for each height in the tree. This adds unnecessary book keeping for the Quad Tree.

Although the number of pairwise collision tests a lower, the amount of references belonging to a single object spans across numerous nodes, and the operations required to do that can slow the algorithm down. This would occur when there is an extremely large amount of elements for a relatively small world. This can be alleviated by setting a larger of amount elements allowed to be in a node.

The final process after populating the Quad Tree is to traverse the tree. Tree traversal is done in post order. After exhausting the recursive call with Traverse, we call the brute force method on the leaf of the tree.

Algorithm 9 Spanning Add Element - Quad Tree

```
1: function SPANNING ADD ELEMENT(e)
2:     if e.MAXX > xOrigin and e.MINY < yOrigin then
3:         QI.ADD ELEMENT(e)
4:     end if
5:     if e.MINX < xOrigin and e.MINY < yOrigin then
6:         QII.ADD ELEMENT(e)
7:     end if
8:     if e.MINX < xOrigin and e.MAXY > yOrigin then
9:         QIII.ADD ELEMENT(e)
10:    end if
11:    if e.MAXX > xOrigin and e.MAXY > yOrigin then
12:        QIV.ADD ELEMENT(e)
13:    end if
14: end function
```

Algorithm 10 Traverse (Post Order) - Quad Tree

```
1: function TRAVERSE(q)
2:     if QI ≠ nil then
3:         TRAVERSETREE(QI)
4:     end if
5:     if QII ≠ nil then
6:         TRAVERSETREE(QII)
7:     end if
8:     if QIII ≠ nil then
9:         TRAVERSETREE(QIII)
10:    end if
11:    if QIV ≠ nil then
12:        TRAVERSETREE(QIV)
13:    end if
14:    BRUTE FORCE(E)
15: end function
```

VI. Optimizations

A. Brute Force

As mentioned earlier the only optimization that can be done with the brute force algorithm is to prune the amount of pairs that are tested for collisions. But for the other algorithms, each of them can benefit from temporal coherence. The current methods provided are the simplest forms to work with, and has no optimizations at all. They must be rebuilt for each call.

If an object in the world changes then the reference to a collection must also change. The most straight forward optimizations to the broad phase collision detection are based off this fact. For all algorithms we can append a flag to each object to check if it has been changed and/or keep a previous state of the object in addition to its current state. If it has been changed, we then reposition its reference within a collection. If it has not been changed then we do not need to rebuild the collection, and only change a subset of objects.

All of this can be done by simply searching the object, deleting it, and reinserting the object into the collection based off of its new position. Some algorithms however can have

specific way to deal with this in certain scenarios.

B. Spatial Partitioning Index

For SPI, the bounds are easily known because of a constant grid size. Referencing neighboring subcollections can be accomplished by simple array arithmetic. For an objects current position we can find the offset to the grid's boundary by modding its position by the grid size. If the change in position is greater than the offset then we can simply translate each reference of an object to its neighboring subcollection. This can be faster than reinserting if the object is large and spans many grids, else reinserting into a single subcollection will always be faster.

As a whole if most objects are moving objects, then rebuilding the entire subcollections would be faster as reinserting requires one hash to find an object and another hash to place it. Whereas rebuilding requires only one hash to place an object.

C. Sweep and Prune

For Sweep and Prune, the straight forward method would be to keep the objects in a heap and extract the minimum when needed. The improved method would make reinserting objects only takes $O(n)$ using insertion sort if objects in the collection are relatively in order. Inserting only a subset of k objects using an insertion sort with binary search would result in a worse case of $O(k \lg(n))$ time, where each binary search would take $O(\lg(n))$ time.

D. Quad Tree

Optimizing a Quad Tree is much trickier. Although the current implementation nodes are stored at the leafs, not all leaves in the Quad Tree are guaranteed to be at the same height. So instead of reinserting at the root which can take $O(\lg(n))$ time to search for the correct node for each element (which is only part of the reinserting process), we can insert based on relative position. Since the nodes are based on bounds and position we can insert references to siblings, and simply translate references across siblings. If sibling is not available then we can go to the cousin by traversing through the parent. This, however is a horrible idea for spanning elements.

VII. Testing

VIII. Conclusion

The conclusion goes here.