To What Extent Are Quad-Tree or Spatial Hashing Algorithms More Efficient for Intersection Testing

1/13/17
Computer Science
3668

Table of Contents

Abstract

From games to complex traffic control systems, collision detection plays an important role in today's modern computers. Currently there are two algorithms – Quad-Tree and Spatial Hashing – that are often used and at times it is based on personal preference. Theoretically Quad-Trees will be less efficient than Spatial Hashing until a certain data size is reached but the choice is ultimately up to the project requirements.

From a broad approach they work as follows, Spatial Hashing is a "bucket" based system where an objects relative area is increased and then inserted into surrounded buckets to be queried against. Quad-Trees are a recursive data structure that divide into quadrants as an object capacity is hit and a bounding box is identified for the selection of a quadrants.

While Spatial Hashing is a flat data structure and does not require lots of computational power, it can provide a lot of false positive results. Quad-Trees, can also return false positives up to a limit to handle different densities and provide a quick and accurate result. The importance is based off the dataset used and how that dataset will be managed.

Word Count: 190

1. Introduction

Over the past decade the scale of games has increased, demanding more from computer programmers in regards to efficiency. Games have gone from classics like Pac-man which has a few hundred sprites and moving objects, to those like Skyrim which are required to handled millions of objects. As games become more intensive and complex, with increasing polygons, running intersection tests within the game becomes nearly impossible and impractical. The solution is to use an algorithm that filters the objects that need to be tested, reducing CPU requirements and decrease processing time. This makes many complex games practical and has allowed for increasing complexity. There are two widely used solutions to handle this: Point Quad-Trees, and Locality-Sensitive Hashing (Spatial Hashing).

The algorithms follow a basic flow of execution. An object – generally a bounding box or a point – will be inserted into the algorithm. These algorithms support re-indexing objects when they are updated but for the sake of evaluating the algorithm for their hit box prediction, this property will be ignored. The same applies for deletion which will also not be considered as it goes beyond the scope. The most important components of these algorithms is their querying ability. What happens within the algorithm during querying is highly dependent on the implementation but generally returns a list of keys or objects that are in proximity of the test point. Several applications use data structures that support three dimensions instead of two – which are really just deviations of the two dimensional data structures. However, for the sake of simplicity it is easier to evaluate the simplest forms of each algorithm. By the end of the execution flow the algorithm should have returned a filtered list of bounding boxes that are potential intersections with the tested area.

There are two implications for doing this comprehensive analysis between the two algorithms. Initially, when a developer makes a game where collisions are a concern they can make an informed decision of what suits their needs the best. For instance, if Quad-Trees are faster under a certain context then it may make sense for them to use Quad-Trees opposed to Spatial Hashing. The use cases go beyond

entertainment purposes. There are scenarios where time is limited and any delay in calculations can have serious implications. In air traffic control systems, where there can be 5000+ airplanes in the air at once predicting a potential crash could be the difference between life and death for hundreds of people. (Daily Mail Reporter, 2012). Consequently, the scope of this application extends beyond current implementations; with the wanted adoption of self-driving cars it becomes a concern how a computer will consider all the traffic. Currently there is about one billion cars in the world and inter-communication across that amount of vehicles is impractical. (Tencer, 2013). However, using one of these algorithms cars could prioritize which vehicles to consider, reducing decision making times.

2. Theoretical Investigation

Before the results from each algorithm can be properly analyzed there must first be an understanding of how they function and the core functionality differences between the two. While both Quad-Trees and Spatial Hashes give the same results have share little in similarity. Consequently, since both algorithms have such vague definition that allows for a multitude of implementations and deviations, understanding the chosen implementation can help provide further context to the efficiency of the test results.

2.1. Quad-Tree

Quad-Trees are a vague data structure which have a multitude of implementations depending on the projects requirements. It is an extension of a binary tree except instead of two pointers it has four pointers. Despite the broad range of implementations, they all follow the same idea: four quadrants broken up in the middle that continue to break up recursively until a max depth is reached. In the context of a Rectangle Quad-Tree each node is divided further only when a max object count is reached and then redistributed among the new quadrants. Each quadrant is always labeled into four different areas: North West (NW), North East (NE), South West (SW) and South East (SE) – depicted by figure 1.

Quad-Tree of 100 Items

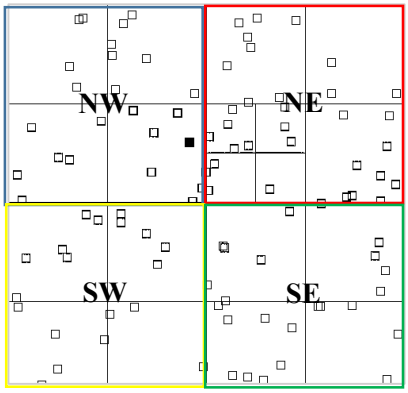

Figure 1

Related Quad-Tree Diagram

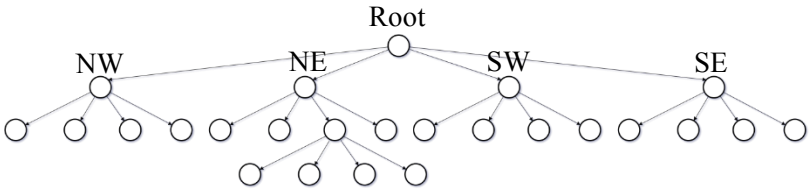

Figure 2

Figure 2 reflects the recursive nature of Figure 1. Each quadrant is divided after a certain intersection threshold is reached. Most of the computational time of a Quad-Tree is spent indexing objects rather than searching as many more tasks can be executed during the splitting or simplification of a Quad-Tree. There are two things that need to be considered when developing a rectangle Quad-Tree. First, what are the maximum amount of quadrants. Second, how many objects does a quadrant divide. The choices made can impact performance and are highly context driven. In the case of this evaluation there are 10 max objects and 12 max subdivisions of quadrants. See appendix 1 for implementation.

2.1.1. Insert

As mentioned previously, Quad-Trees have a recursive structure making a lot of their functions dependent on the depth and follow the same design pattern. Furthermore, because a Quad-Tree splitting is contingent on the amount of objects per quadrant the efficiency can be significantly impacted. In the best case scenario, insertion yields $O(n \log n)$ performance; however, if the algorithm splits quadrant inefficiently, it can be as slow as $O(n^2)$. The process flow of a Quad-Tree goes as following:

- First the bounding is broken up into two points defined as: $p1$(upper-left corner) and $p2$ (lower-right corner)
- Then each quadrant ($Q = [X_{1q}: X_{2q}] \times [Y_{1q}: Y_{2q}]$) is calculated for each point:

$$x_{mid} = (X_{1q} + X_{2q})/2 \quad y_{mid} = (Y_{1q} + Y_{2q})/2$$
$$Q_{NE} = \{p_x > x_{mid} \wedge p_y > y_{mid}\}$$
$$Q_{NW} = \{p_x \leq x_{mid} \wedge p_y > y_{mid}\}$$
$$Q_{SW} = \{p_x \leq x_{mid} \wedge p_y \geq y_{mid}\}$$
$$Q_{SW} = \{p_x > x_{mid} \wedge p_y \leq y_{mid}\}$$

There are many benefits and limitations to this structure. The first benefit is the data structure is not very tall. Therefore, when the Quad-Tree re-indexes – redistributing boxes to fit the newly split quadrants – it only handles 10-20 objects instead of hundreds. When the tree is only operating on a few objects a lot of CPU processing time is saved. However, the data stored is larger than alternative structures and checking which bounding-box intersects with a quadrant uses more ticks and affects the time used splitting a section. Since Quad-Trees are a recursive data structure one quadrant splitting increases its memory requirements by a multiple of four; consequently, its alternative has a flat structure. Furthermore, the operation above shows the amount of comparisons required for a singular point in testing quadrant intersection. The math above only considers one point whereas a rectangle is made-up of two points and then tested against about ten rectangles totaling twenty plus operations for the division of a quadrant.

2.1.2. Query

The primary benefit of Quad-Trees is their efficiency regarding querying, as most of the computation time is spent distributing objects across different quadrants. When looking at the relative efficiency the amount of queries is equivalent to $O(\frac{4n}{10})$ when n represents the total amount of nodes within the tree. The benefit of this algorithm is that when there is less objects within a Quad-Tree the less computation time it takes; additionally, as the Quad-Tree grows the amount of processing time does not increase significantly. There are also cases where efficiency could go up to $O(1)$ if the Quad-Tree has yet to split yet or if the quadrant being queried has only been split one time. This is because Quad-Trees are not balanced and the query time is highly dependent on where the object being queried is located.

Take for instance *Figure 3* after the query completes its regression it has reached three sublevels requiring a total computation time of $O(12)$. In the figure below the blue represents the area intersected is represented by a blue rectangle and then the parent quadrant is represented by a faint grey.

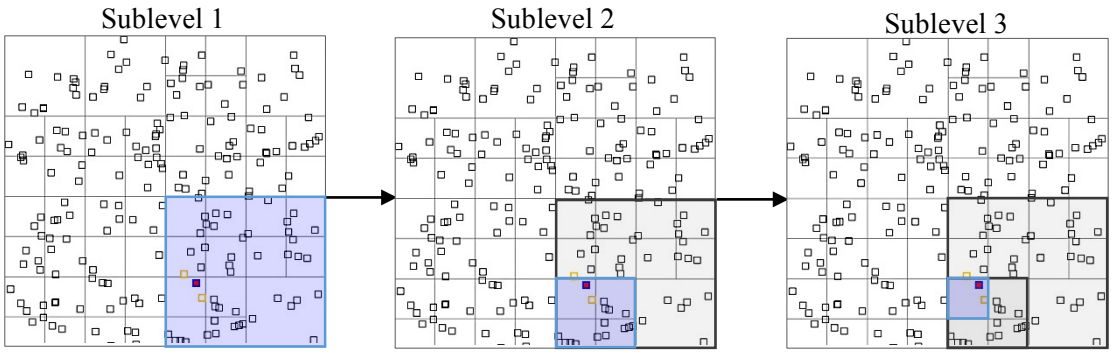

Figure 3

The benefits to Quad-Tree hitbox detection is the speed of testing, especially when there are lots of objects.

2.2. Spatial Hashing

Spatial Hashing works by increasing the total area of the bounding boxes being tested; while, retaining the possible intersections. It increases the probability of detecting intersections by grouping objects into "buckets" and then testing which objects buckets are contained. These buckets are created in the forms of hashes by taking all potential points of the reduced object and associating it into a map where the queried object is tested against. The hashing function is depicted as following:

$$(x \gg shift < xSH < x + width \gg shift) \cap (y \gg shift < ySH < y + width \gg shift)$$

The algorithm depends on a right shift binary operator and works by taking the left operand and moving it right by the shift value specified, equating the value to $\left\lfloor \frac{x}{2^{shift}} \right\rfloor$. The area of the bounding box is increased by a multitude of 2^{shift} but remaining in the same position; however, because the area is so small each individual pixel can be considered a bucket and the corresponding boxes can be stored. This grouping method is depicted by figure 4. The green represents what buckets the red test point is categorized into while the yellow boxes are considered "predicted intersections". It could be misconceived that the yellow boxes need to be contained in the green buckets; however, the overall area is being increased and therefore the yellow boxes are contained within but not visualized to equate the growth. Another attribute present is that not all

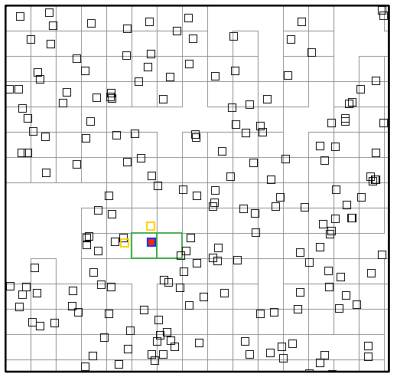

Figure 4

"buckets" exist yet because there is not a point to occupy the bucket and as a consequence would not store the bucket in memory. This differs from Broad-Phase collision detection because instead of diving an area into a grid and testing if an objects bounding box intersects with any of the cells it tests if hashes are equal. In retrospect, the two algorithms are not too different but in regards to this evaluation Broad-Phase collision is not compared. See appendix 2 for implementation.

2.2.1. Insertion

Due to the simplicity of the structure there is not much to insertion or querying beyond the hashing function. The only performance dependent structure used within insertion is a Hash Table. They work by taking an associative array and then mapping keys to values and matching object indexes. In the worst case scenario an insertion into a Hash Table can take O(n) but on average O(1) is the algorithm efficiency. In a well-constructed table, the number of instructions required to add an object should not be dependent on the quantity of objects in the Table. Figure 5 depicts the relationship behind a Hash Table used for storing the objects on the map. When an object is inserted it can occupy multiple buckets and depending on if the bucket exists or not it will either create a bucket or append itself to the existing array.

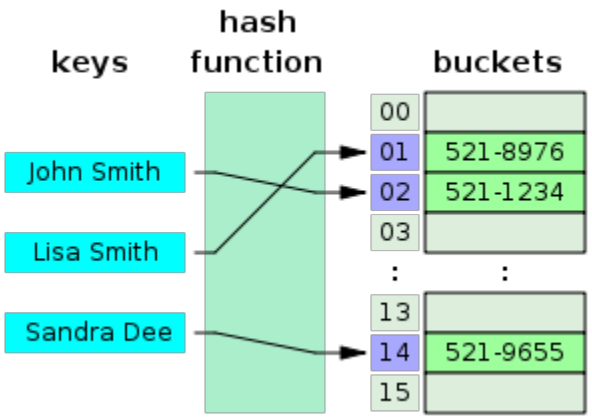

Figure 5

2.2.2. Query

Querying is very similar to insertion except for two small aspects. First, the object queried must be re-hashed to be able to match the hash with the value. Additionally, the hash queries multiple different keys because of the design of the hashing function resulting in multiple different "hashes". The hashes in this scenario are not intended to be unique from other objects in the map but rather act as a way of labeling "buckets". On average the performance expected would be O(1) but depending on the Hash Table used performance can be O(n). There is a very strict balance between accuracy and the *shift* value used to obtain the maximum performance. Increasing the shift reduces the amount of buckets but will

return more false positives. Decreasing the shift will increase buckets and reduce the amount of false positives but exponentially increase ram usage.

3. Experimentation and Analysis

3.1. Methodology

Testing was straight forward but there were problems with accuracy induced through numerous factors. First, the data was computed using Java's native *java.util.Random* to ensure that the tests were in multiple different locations and not influenced by placement so that the data can be a pure reflection of algorithm efficiency. Only test points would be randomized while object size were kept to 10x10 pixels and the canvas was kept to 500x500 pixels. Without keeping constants, it would be hard to evaluate the factors that influenced the algorithms; furthermore, the investigation focuses on bulk data and not type of data. The quantity of objects would increase from 1 to 1000 with 5,000 collision tests run per quantity. While, 1 to a 1000 may seem small the density was high which would be the "worst" case scenario in intersection testing.

The 5,000 test quantity is an anecdote to a problem with Java's speeds and thread priority. Sometimes, Java ran garbage collection or recompiled code which caused a very minute impact on performance overall but still impacted test results. By running 5000 tests JVM induced lag can be averaged out. Another preventative measure is running garbage collection after every test and only running one type of test – Spatial Hash or Quad-Tree – per execution. This prevents ram clipping which is when the JVM attempts to create more heap space storage and holds the thread. The results are then exported to a *results.csv* file and imported into excel for analysis. To ensure maximum quality both tests are running on the same machine, with no other nonessential programs, with the same data set, and executing at the highest thread priority. Finally, a visualizer was made to show how the algorithm responds to different scenarios and depict the actions of the algorithm along with the overall results.

3.2. Data and Results

3.2.1. Performance

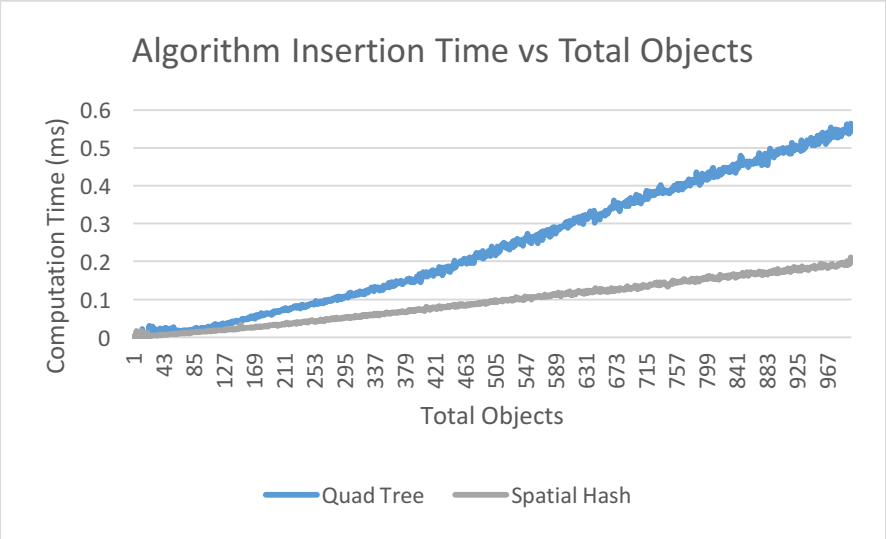

Figure 6

The results from this test were expected after analyzing the potential performance through Big-O notation. The Quad-Tree shows that the performance has an $O(n^2)$ growth – despite the exponential growth being slow; meanwhile, the Spatial Hash had a O(n) performance. Unlike, other tests insertion took the longest computational time than any other type of function. Therefore, one could assign the most importance of their intersection algorithm on time it takes to insert but there are other factors that needs to be considered. For example, many systems do not completely "re-insert" after every execution but rather update the structure to move an object to a different position; which is more efficient when not all objects are moving. This was not evaluated in this paper but could have validity in a practical example. Consequently, performance is not everything and the reason that these algorithms are integrated is because running real intersections are more resource intensive than the algorithms being evaluated.

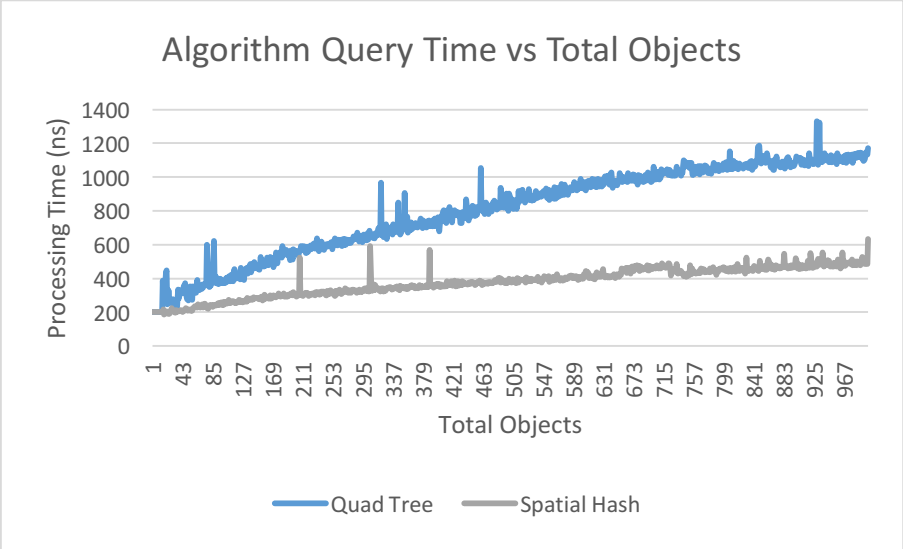

Figure 7

Again Spatial Hashing is superior in computation time as it following a linear trend of O(n) and the Quad-Tree following an exponential curve. However, it is important to understand the times measured are in nanoseconds – a billionth of a second. This makes it hard to classify the speed of the Algorithms because tasks such as reading the system clock time could have impacted the algorithms speed. Since the Quad-Tree is on an exponential curve as it approaches its "asymptote" the line will stabilize and the performance differences in the Spatial Hash will cause it to take longer than the Quad-Tree. After finding the lines of best fit for the Quad-Tree at $y = 70.614x^{0.3993}$ with an R^2 of 0.9462 and Spatial Hash at $y = 0.2798x + 237.55$ with an R^2 of 0.9185 it would be more efficient to use Quad-Tree for queries beyond about 8500 items. Although this is only an extrapolation of the data so results may differ.

3.2.1. Ram Usage

The one limitation in measuring efficiency was the inability to measure ram usage. The heap is often filled with many different java objects and garbage collection runs so fast that by the time the code tried to measure ram usage it had already been collected. While it is difficult to judge, one could claim that the current design structure of Spatial Hashing would take more ram space than a Quad-Tree. Spatial Hashing uses a Hash Map to store data which means that references to objects are stored inside the Map in a list that gets longer as more objects are added; therefore, the depth of Spatial Hash could cause a lot

of ram usage. Consequently, Quad-Trees are stored as objects recursively which means that the ram of each object could exceed the amount of a Spatial Hash. Either side has a compelling argument and although there is no data to validate these claims it is important to understand the theoretical performance differences.

3.2.1. Quality of Results

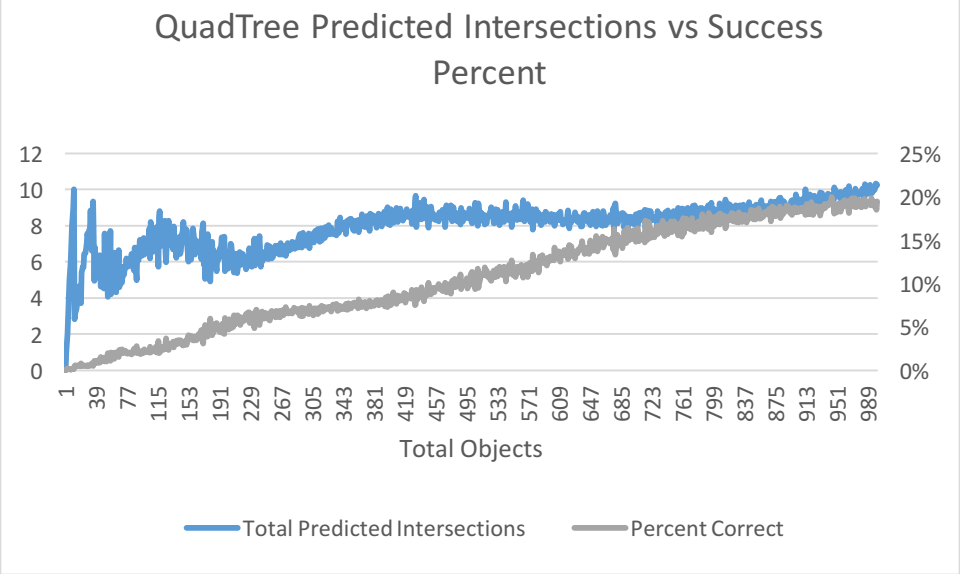

Figure 8

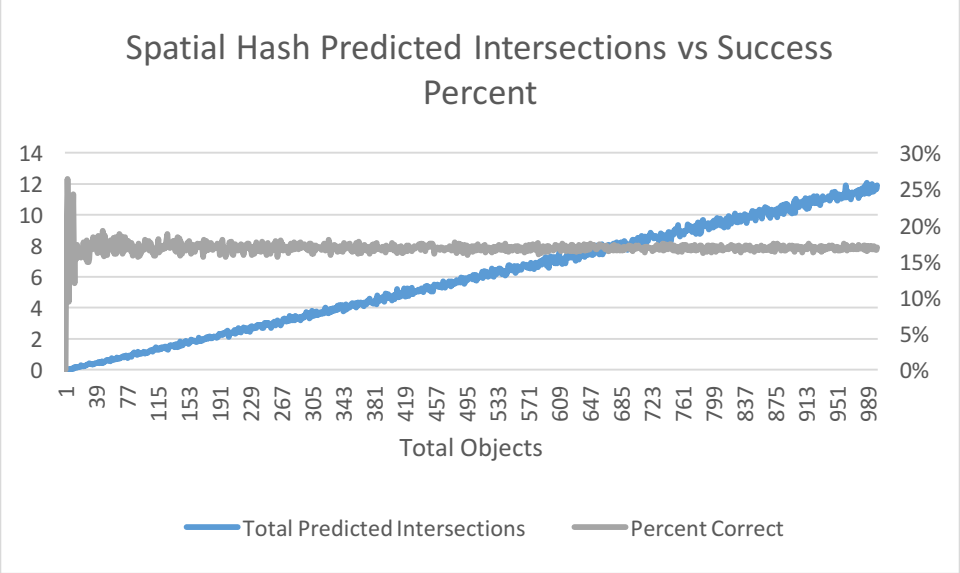

Figure 9

The results of Figure 8 and Figure 9 need to be analyzed together because they both show the relationship between results and “noise” or false positives. At around 700 objects – when there is a lot of

cluttering and many objects in a dense space – the results of Spatial Hashing becoming less accurate with a 15% success rate while Quad-Trees approach a 20% success rate. With both Spatial Hashing and Quad-Trees, the accuracy of intersections will decrease as density increases because more objects will occupy the quadrants or buckets. Quad-Trees are affected less by this because they recursively divide to increase accuracy but Spatial Hashing as a limited amount of buckets which results in the decrease of accuracy as density increases. It is expected that Spatial Hashing's accuracy to continue decreasing while the Quad-Trees accuracy will eventually plateau.

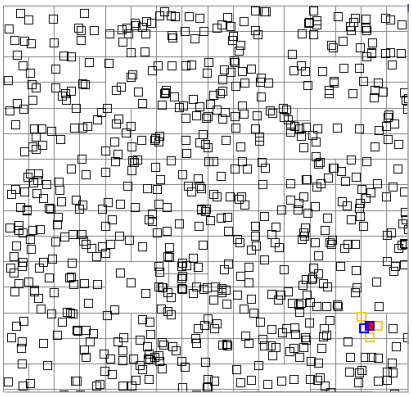

Figure 10 (Quad-Tree)

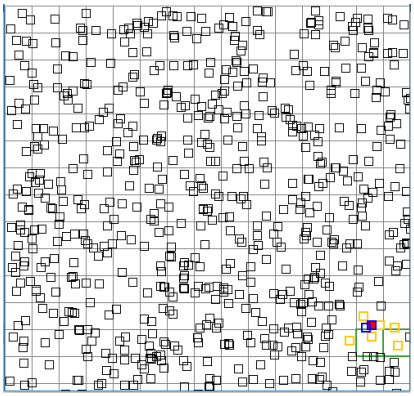

Figure 11 (Spatial Hashing)

As evidenced by a test where there are 700 objects on a 500x500 plane the Spatial Hashing technique of decreasing size to increase area causes a lot of false positives; meanwhile, Quad-Trees recursive divide only returns the objects the test point resides. This is a very important factor when considering an algorithm because the time it takes to test an intersecting of a complex shape can often take a significant amount of more time and a 5% difference after several calculations can have a significant impact.

3.2.2. Visual Analysis

Example 1 ~ Small Datasets

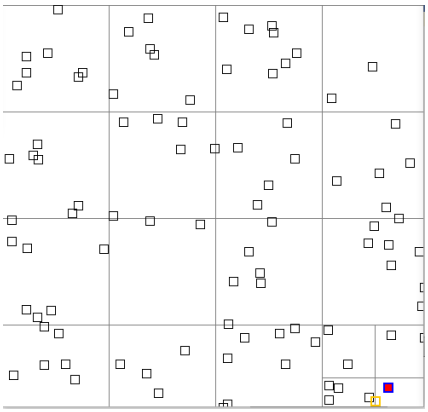

Figure 10 (Quad-Tree)

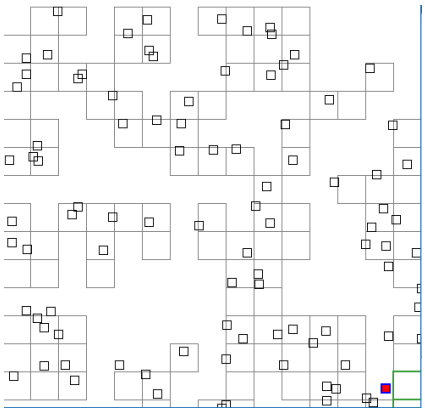

Figure 11 (Spatial Hashing)

When using a Quad-Tree, there will always be at least one object because Quad-Trees return the entire dataset of the split cell and it requires there to be more than a set amount of objects for a split to occur. However, with Spatial Hashing the objects depend on groupings or "buckets. This emphasizes that the Quad-Trees are more effective at larger datasets rather than smaller and it would be smarter to use Spatial Hashing; ignoring the fact that an implementer would most likely not need a collision system at a scale this small. The other aspect to consider is that Spatial Hashing has a direct correlation to the size of canvas while Quad-Trees are independent of canvas size in regards to data capacity requirements.

Example 2 ~ Large Datasets

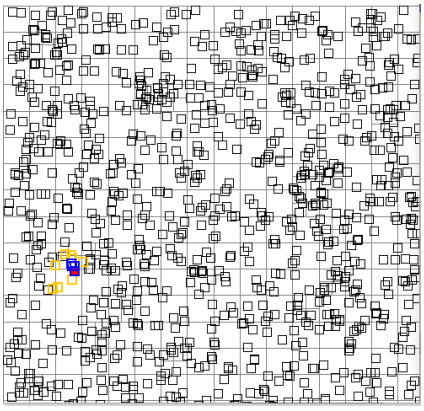

Figure 12 (Quad-Tree)

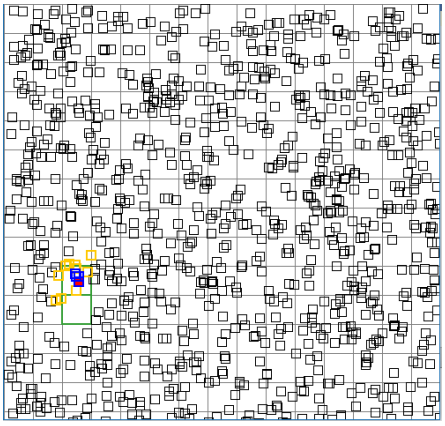

Figure 13 (Spatial Hashing)

This is where Quad-Trees can out compute Spatial Hashing. While, Quad-Trees had spent a majority of their time computing the divisions of the tree the more accurate form of hitbox detection

allows for a less noisy result. Furthermore, if these objects were moving a Spatial Hash would have to search through a list of a single quadrant and then rehash while a Quad-Tree would have to move one level in the hierarchy and redo its bounding test. If this were to be classified into big-o notation it would be: O(1) for Quad-Trees and O(n) for Spatial Hashing. Therefore, despite not being specifically analyzed, for certain use cases Quad-Trees are more efficient than Spatial Hashing.

4. Conclusion and Reflection

4.1. Limitations of Investigation

There are aspects that could be considered in the future to make the evaluation more expansive. For instance, only the insertion and querying function were tested when functions like update – when the object is moved in the data structure – could be evaluated to see how the structure responds to a more realistic scenario. Deletion was also not tested which could have had an impact on the evaluation as both delete in different ways. Furthermore, data in regards to ram performance were unable to be evaluated which could have shown which could cause more performance dampening in the form of frequent garbage collection or even halting if there is not enough ram available.

The algorithms tested and environment could have also of been changed to gain a more accurate representation of the data. For instance, different sizes of the shapes, canvas size, and types of shapes could have been used to provide variation of the dataset and show how each mythology handles more complex scenarios. This was a limitation of time and resources that was not necessary but could show the more practical use cases. Finally, Broad-Phase could also be evaluated which would be the final algorithm to analyze out of all three major hit box detection structures.

4.2. Conclusion

Overall, while the efficiency of Quad-Trees is less than Spatial Hashing until it approaches 700 objects its increased accuracy in addition to its ability to handle querying would make it faster for large scale applications. The theoretical results paralleled the experimental results confirming what was to be expected from the tests. This emphasizes that for smaller scale uses Spatial Hashing is the better option to

use for its performance and predictable response times. The problem is that which algorithm to use is highly dependent on the use case. If the dataset has lots of updates and queries with a high density Quad-Trees are better suited because of their ability to limit the clustering of data. If the data has lots of insertion and deletion Spatial Hashes are direct correlations to the dataset so it would be less ram and more efficient to use.

In conclusion, if the test set is smaller or requires lots of insertion then use Spatial Hashing; however, if the dataset is queried often and is expansive than use Quad-Trees. When comparing overall efficiency to the speed the best choice would be Quad-Trees for its reliable results which is more important as bounding box tests require more time than the algorithms tests.

Works Cited

DAILY MAIL REPORTER. (2012, September 17). Rush hour in the skies: Real time map that shows you every plane in the air right now [Newsgroup post]. Retrieved from Dailymail website: http://www.dailymail.co.uk/news/article-2204838/Rush-hour-skies-Real-time-map-shows-plane-air-right-now.html

Tencer, D. (2013, February 19). Number Of Cars Worldwide Surpasses 1 Billion; Can The World Handle This Many Wheels? [Newsgroup post]. Retrieved from Huffington Post website: http://www.huffingtonpost.ca/2011/08/23/car-population_n_934291.html

Works Consulted

Alban, M. (2008, February 22). *Locality-Sensitive Hashing* [PowerPoint slides]. Retrieved January 10, 2017, from Visual Recognition and Search website: http://www.cs.utexas.edu/~grauman/courses/spring2008/slides/Marc_Demo.pdf

Kingsford, C. (2007, August 30). *Quad-Trees* [Lecture notes]. Retrieved October 6, 2016, from Data Structures website: https://www.cs.umd.edu/class/spring2008/cmsc420/L17-18.QuadTrees.pdf

Lu, Y. (2014, April 23). *Quadtrees* [PowerPoint slides]. Retrieved April 7, 2016, from http://web.eecs.utk.edu/~cphillip/cs594_spring2014/quadtree-Allan.pdf

Mehta, D. P., & Sahni, S. (2005). *Handbook of data structures and applications* [pdf]. Retrieved from http://www.e-reading.club/bookreader.php/138822/Mehta_-_Handbook_of_Data_Structures_and_Applications.pdf

Paulevé, L., Jégou, H., & Amsaleg, L. (2010). Locality sensitive hashing: a comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, *31*(11), 1348-1358. http://dx.doi.org/10.1016/j.patrec.2010.04.004

Samet, H., & Webber, R. E. (1985). Storing a Collection of Polygons Using Quadtrees. *ACM Trans. Graph*, (4). http://dx.doi.org/10.1145/282957.282966

Tsai, Y. H., & Yang, M. H. (2014). Locality preserving hashing. *IEEE International Conference on Image Processing*, 2988-2992. http://dx.doi.org/10.1109/ICIP.2014.7025604

Appendix

Appendix 1 – Quad-Tree Algorithm

```
public class QuadTree extends Rectangle implements IntersectionAlgorithm {

   @Setter
   private int MAX_OBJECTS = 10;
   @Setter
   private int MAX_LEVELS = 13;

   private int level;
   private List<Rectangle> objects;
   @Getter
   private QuadTree[] nodes;

   public QuadTree(int width, int height) {
      this(0, 0, 0, width, height);
   }

   private QuadTree(int pLevel, int x, int y, int width, int height) {
      super(x, y, width, height);
      this.level = pLevel;
      this.objects = new ArrayList<>();
      this.nodes = new QuadTree[4];
   }

   @Override
   public void clear() {
      objects.clear();

      for (int i = 0; i < nodes.length; i++) {
         if (nodes[i] != null) {
            nodes[i].clear();
            nodes[i] = null;
         }
      }
   }

   @Override
   public void insert(Rectangle box) {
      if (isSplit()) {
         Integer[] indexs = getIndexs(box);

         if (indexs[0] != -1) {
            Stream.of(indexs).forEach(i -> nodes[i].insert(box));

            return;
         }
      }

      objects.add(box);

      if (objects.size() > MAX_OBJECTS && level < MAX_LEVELS) {
         if (!isSplit()) {
            split();
         }

         int i = 0;
         while (i < objects.size()) { //Reindex all objects
```

```
            Integer[] indexs = getIndexs(objects.get(i));
            if (indexs[0] != -1) {
               Rectangle boxToRemove = objects.remove(i);
               Stream.of(indexs).forEach(x -> nodes[x].insert(boxToRemove));
            } else {
               i++;
            }
         }
      }
   }

   @Override
   public List<Rectangle> retrieve(Rectangle box) {
      return this.retrieve(box, new ArrayList<>());
   }

   private List<Rectangle> retrieve(Rectangle box, List<Rectangle> returns) {
      Integer[] indexs = getIndexs(box);
      if (indexs[0] != -1 && isSplit()) {
         Stream.of(indexs).forEach(i -> nodes[i].retrieve(box, returns));
      }

      returns.addAll(objects);

      return returns;
   }


   private void split() {
      int subWidth = (int) Math.floor(width / 2f), subWidthOffset = (int)
Math.ceil(width / 2f);
      int subHeight = (int) Math.floor(height / 2f), subHeightOffset = (int)
Math.ceil(height / 2f);

      nodes[0] = new QuadTree(level + 1, x + subWidth, y, subWidthOffset, subHeight);
      nodes[1] = new QuadTree(level + 1, x, y, subWidth, subHeight);
      nodes[2] = new QuadTree(level + 1, x, y + subHeight, subWidth, subHeightOffset);
      nodes[3] = new QuadTree(level + 1, x + subWidth, y + subHeight, subWidthOffset,
subHeightOffset);
   }

   private boolean isSplit() {
      return nodes[0] != null;
   }


   private Integer[] getIndexs(Rectangle pRect) {
      if (!isSplit()) return new Integer[]{-1};
      List<Integer> indexs = new ArrayList<>();
      if (nodes[0].intersects(pRect) || nodes[0].contains(pRect)) indexs.add(0);
      if (nodes[1].intersects(pRect) || nodes[1].contains(pRect)) indexs.add(1);
      if (nodes[2].intersects(pRect) || nodes[2].contains(pRect)) indexs.add(2);
      if (nodes[3].intersects(pRect) || nodes[3].contains(pRect)) indexs.add(3);

      if (indexs.size() <= 0) {
         System.out.println("Help");
         return new Integer[]{-1};
      }

      return indexs.toArray(new Integer[indexs.size()]);
```

```
    }

    public List<Rectangle> getInnerBoxes() {
        return this.getInnerBoxes(new ArrayList<>());
    }

    private List<Rectangle> getInnerBoxes(List<Rectangle> boxes) {
        if (level != 0 || Stream.of(nodes).allMatch(x -> x == null)) {
            boxes.add(new Rectangle(x, y, width, height));
        }
        for (QuadTree tree : nodes) {
            if (tree == null) {
                continue;
            }
            boxes.addAll(tree.getInnerBoxes());
        }
        return boxes;
    }
}
```

Appendix 2 – Spatial Hash Algorithm

```
public class SpatialHash implements IntersectionAlgorithm {

    public final static int DEFAULT_POWER_OF_TOW = 5;

    private int shift;
    private Map<String, ArrayList<Rectangle>> hash;

    public SpatialHash() {
        this(DEFAULT_POWER_OF_TOW);
    }

    public SpatialHash(int powerOfTwo) {
        this.shift = powerOfTwo;
        this.hash = new HashMap<>();
    }

    @Override
    public void insert(Rectangle box) {
        List<String> keys = this.getHash(box);
        for (String key : keys) {
            if (this.hash.containsKey(key)) {
                this.hash.get(key).add(box);
            } else {
                this.hash.put(key, new ArrayList<>(Collections.singletonList(box)));
            }
        }
    }

    @Override
    public List<Rectangle> retrieve(Rectangle box) {
        List<Rectangle> results = new ArrayList<>();
        this.getHash(box).stream().filter(key -> this.hash.containsKey(key)).forEach(key
-> results.addAll(this.hash.get(key)));
        return results;
    }

    @Override
    public void clear() {
        this.hash.clear();
```

```
   }

   public List<String> getHash(Rectangle box) {
      int sx = (int) box.getX() >> shift,
            sy = (int) box.getY() >> shift,
            ex = (int) (box.getX() + box.getWidth()) >> shift,
            ey = (int) (box.getY() + box.getHeight()) >> shift;
      List<String> keys = new ArrayList<>();
      for (int y = sy; y <= ey; y++) {
         for (int x = sx; x <= ex; x++) {
            keys.add(x + ":" + y);
         }
      }
      return keys;
   }
}
```

Figure 5: Stolfi, J. (2009, April 10). *Hash Table* [Photograph]. Retrieved from https://commons.wikimedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg