## Extended essay cover

Candidates must complete this page and then give this cover and their final version of the extended essay to their supervisor.

<table>
<thead>
<tr>
<th>Candidate session number</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Candidate name</td>
<td></td>
</tr>
<tr>
<td>School name</td>
<td></td>
</tr>
<tr>
<td>Examination session (May or November)</td>
<td>May</td>
</tr>
</tbody>
</table>

Diploma Programme subject in which this extended essay is registered: **Computer Science**

(For an extended essay in the area of languages, state the language and whether it is group 1 or group 2.)

Title of the extended essay: **The Effectiveness of Combining Sweep and Prune, and Spatial Hashing**

### Candidate’s declaration

This declaration must be signed by the candidate; otherwise a mark of zero will be issued.

The extended essay I am submitting is my own work (apart from guidance allowed by the International Baccalaureate).

I have acknowledged each use of the words, graphics or ideas of another person, whether written, oral or visual.

I am aware that the word limit for all extended essays is 4000 words and that examiners are not required to read beyond this limit.

This is the final version of my extended essay.

Candidate’s signature: ____________________________ Date: 21/5/2015
visor's report and declaration

Supervisor must complete this report, sign the declaration and then give the final version of the extended essay, with this cover attached, to the Diploma Programme coordinator.

Name of supervisor (CAPITAL letters) __________________________

Comment, as appropriate, on the candidate's performance, the context in which the candidate undertook research for the extended essay, any difficulties encountered and how these were overcome (see page 13 of extended essay guide). The concluding interview (viva voce) may provide useful information. These points can help the examiner award a level for criterion K (holistic judgment). Do not comment on any personal circumstances that may have affected the candidate. If the amount of time spent with the candidate was zero, you must explain this, in particular how it was then possible to authenticate the essay as the candidate’s own work. You may attach an additional sheet if there is insufficient space here.

Context of the research question is clearly demonstrated. Some appropriate sources have been consulted and relevant material has been selected. A lot of data has been collected using the candidate’s computer program. The essay in general demonstrates a good understanding and knowledge of the topic studied. Ideas are presented in a logical and coherent manner, and a sound argument is developed. The essay shows some application of appropriate analytical skills, which is considerably effective. The candidate used the terms relevant to the subject. An effective conclusion is clearly stated and some evidence presented. It also includes future work or some resolved questions in relation to the topic. The formal presentation is satisfactory, although there is some room for improvement including layout, organization and other formal elements. The candidate also included a clearly stated abstract addressing the research question that was investigated.

I read the final version of the extended essay that will be submitted to the examiner.

To the best of my knowledge, the extended essay is the authentic work of the candidate.

The section entitled “Responsibilities of the Supervisor” in the EE guide states that the number of contact hours with candidates is between 3 and 5 hours. Schools will be contacted if, or where 0 hours are stated and there lacks an explanation. Schools must record the number of hours spent is significantly excessive compared to the recommended number.

3 hours with the candidate discussing the progress of the extended essay.

Supervisor’s signature: __________________________ Date: 02/25/2015

Declaration must be signed by the supervisor; otherwise a mark of zero will be issued.
Extended Essay in
Computer Science Group 4

The Effectiveness of Combining Sweep and Prune, and Spatial Hashing

An investigation of the efficiency of a hybrid of two broad-phase collision detection algorithms

Session:
May 2015

Word Count:
3879
Abstract

The broad-phase is an important part of collision detection as it is responsible for narrowing down pairs of objects that could potentially be colliding. This investigation combines two established broad-phase collision detection methods, sweep and prune, and spatial hashing, and quantifies the effectiveness of the hybrid method relative to its individual counterparts. The effectiveness of this hybrid method is compared to the other methods in two circumstances; virtual environments with large amounts of objects and virtual environments with objects moving at high speeds. This is done by implementing all three methods in a Java simulation and collecting average runtime data of collision detection. It was found that the hybrid method is more efficient and deteriorates at a slower rate than the other two in cases with large amounts of objects and in cases with moderate object speeds. Spatial hashing was found to be preferable at high object speeds.

Word count: 148
1 Introduction

Collision detection is an important issue that has applications in both physical environments, such as robotics, and virtual environments, such as video games. Thus, its optimization is important for the efficient function of many systems. Collision detection is often divided into two phases: the broad-phase and the narrow-phase. (Tracey, Buss, & Woods, 2009). The broad-phase, specifically with the regards to virtual environments, is the focus of this paper. The broad-phase seeks to lower the amount of collision tests that have to be performed by reporting only object pairs that could reasonably be colliding. Two objects on opposite sides of a virtual world, for example, cannot reasonably collide. Thus, the pair would not be reported by the broad-phase algorithm. The narrow-phase is responsible for the subsequent testing to indicate whether or not two objects do in fact collide. The broad-phase is particularly important to collision detection in environments with large amounts of objects. Without checking for objects that could reasonably be colliding, $n^2$ collision tests would have to be processed, where $n$ is the number of objects in the environment (Luque, Comba, & Carla, 2005). This processing would require significant system resources.

Two methods that are currently used to handle broad-phase collision detection are sweep and prune, and spatial hashing. This paper investigates how significantly a hybrid of these two methods would improve efficiency over its individual components in a large scale 2D virtual environment. Scale in this paper is defined by two measures, the amount of objects in the virtual environment and the speed of the moving objects. Increasing both of these measures can lead to the deterioration of a broad-phase algorithm and thus, two investigations were carried out and analyzed. A similar investigation (combining sweep and prune and another broad-phase algorithm) inspired this investigation (Tracey, Buss, & Woods, 2009). It was, however, done in the context of another investigation and focused only on object amount, without a focus on object speed.

2 Simulation Design

In order to compare the efficiency of the two broad-phase algorithms and the hybrid method, a simulation was written in Java code which implements the three algorithms and collects relevant data.

2.1 Explanation and Implementation of Techniques

In general, the implementation of broad-phase collision detection techniques makes use of bounding volumes as an approximation of the objects' size. This is because only potential collisions are reported and thus, accuracy is not imperative. In this simulation, a square (AABB box) around the objects is used as an approximation of their bounds. The different techniques are then implemented based on these boxes.

Although spatial hashing and sweep and prune follow generally prescribed algorithms, there are small optimizations and variances in their implementation that can affect the data collected in this experiment. The implementations used attempt to reach the optimal level of performance for the simulation environment at hand and are described below.
2.1.1 Sweep and Prune

The algorithm for sweep and prune is based on an older broad-phase algorithm known as the sweeping plane. The sweeping plane involves maintaining a list of the minimums and maximums of each object's bounding box for each Cartesian axis (one for each dimension). After the list is sorted, a pass is done that stores the objects whose minimums have been reached and whose maximums have not (Tracey, Buss, & Woods, 2009). If another object’s extreme is reached before the first object’s maximum has, the two objects' bounding volumes overlap and therefore the pair is reported as a potential collision.

The key to sweep and prune that increases its efficiency over the sweeping plane is its exploitation of temporal coherence. Temporal coherence is the state where objects do not move significantly between cycles. Between cycles, the lists of bounds only slightly changes. This allows for an extremely efficient insertion sort (Tracey, Buss, & Woods, 2009). The storing of potential collisions is also done simultaneously during the insertion sort, further increasing efficiency.

In this implementation, at every swap of a minimum or maximum, the bounding boxes of those minima or maxima were checked for overlap. If the bounding boxes overlapped, the objects approximated by those bounding boxes were reported as a potential collision pair. A different method of implementation would be to maintain a triangular matrix, with a spot for each pair of objects in the simulation (Cohen, Lin, Manocha, & Madhav, 1995). The triangular matrix would contain two Booleans indicating whether or not the bonding boxes overlapped in the two dimensions. As the sorts of the two lists are done, a Boolean is turned to true if the objects overlap in a particular dimension. Two trues indicate that the bounding boxes overlap in both dimensions and so they are reported as a potential collision pair. Although this method slightly reduces the tasks that have to be carried out (since a check for bounding box overlap does not have to be carried out at each swap), it requires an extra amount of memory that will experience quadratic growth with the amount of objects in the virtual environment.

2.1.2 Spatial Hashing

Spatial hashing involves the division of the virtual environment into equal sized cells. The objects are then placed in the correct cells based on their bounding volumes (Teschner, Heidelberger, Muller, Pomeranets, & Gross, 2003). The components of the bottom right vertex are divided by the cell size then rounded down to give the cell location. Using the calculated locations, the objects are hashed into an index of cells. Only objects in the same cells are checked for overlap.

When implementing this technique, the optimal grid size has to be found. If the cell size is too large, the amount of potential checks would be large because there would be many objects in each cell. If the cell size is too small, a single object would occupy multiple cells. This would once again increase the amount of tasks that would have to be processed (more insertions per object). A simple investigation was done to find the optimal cell size, by increasing cell length as multiples of the object radius and keeping other factors constant. 1000 objects were kept in the virtual environment and the objects moved at a speed of 1 unit frame⁻¹ (unit per frame).
It can be seen in figure 1 that the optimal cell size is from five to fifteen times the radius of the objects. Thus, spatial hashing was implemented in this simulation with a cell size of ten times the radius objects.

The typical implementation of spatial hashing differs from sweep and prune because it does not utilize temporal coherence. In every cycle update, the locations of the objects have to be found and rehashed into the correct indices.

2.1.3 Hybrid Method

A combination of the two aforementioned methods would involve running a sweep and prune over a limited area of the space, one of the cells that the virtual environment is divided to in the spatial hashing algorithm. The premise for this idea is that a sweep and prune over a small area of space will reduce the amount of objects overlapping on only one axis. In the sweep and prune algorithm, when the objects' bounds are being sorted, if the bounds of two objects swap in just one of the lists a test will be carried out to see if the AABB bounding boxes overlap on both. This would increase the tasks that need to be processed. Similar to spatial hashing, a cell size with optimal performance has to be found for the implementation of the hybrid method. The test was run with the same parameters.
The size of the virtual environment is 5000 units, so the smallest possible cell length would be 2500 units (as they all have a uniform size). As the cell size decreases, the average runtime of the algorithm decreases. This is likely because a greater amount of cells would require more runs of the sweep and prune algorithm (insertion sorts). There is an optimal level where the extra runs are counteracted by the decrease in overlaps on one axis. In the previously mentioned external test of the hybrid method, the optimal object amount per cell was found to be 300 (Tracey, Buss, & Woods, 2009). This is close to what was found in this experiment as there would be an average of 250 objects per cell.

The hybrid method cannot rehash the objects at every cycle like spatial hashing because it will no longer be exploiting temporal coherence. This would significantly take away from the efficiency of the sweep and prune algorithm. Thus, a method for handling movement of objects from cell to cell has to be implemented. The java code for the algorithm used to handle cell migration is located below:

```java
for (int i=0; i<xAxis.length; i++)
{
  // removes vertices who have left the area (less than the minimum x)
  if (xAxis[i].getValue() < minx)
  {
    xAxis2.remove(xAxis[i]);
    xAxis2.remove(xAxis[i].getOwner().getMinX());
    xAxis2.remove(xAxis[i].getOwner().getMaxY());
    yAxis2.remove(xAxis[i].getOwner().getMinY());
  }
  else
  {
    // reinserts the circle in the correct cell
    BroadPhaseSimulation.hybrid.register(xAxis[i].getOwner());
  }
}
```
This implementation works by traversing each of the two lists of extremes \( (x\text{Axis2} \text{ and } y\text{Axis2}) \) both forwards and backwards. In the case above the list of x-coordinates is traversed forwards, so the points are checked to see if they are less than the minimum bound of the cell. If they are, the objects' coordinates are removed and the object is inserted in the right cell. If a point in the traversal is greater than the cell's minimum bound, the loop can stop because the list of points is sorted in ascending order and all points thereafter would be greater than the cell's minimum. In order to increase efficiency of the traversal, when an object is inserted into the correct cell, if it was closer to the minimum bound it was inserted at the front, if it was closer to the maximum bound it was stored at the back. This was handled by the method `register(Circle circle)`, called on line 21.

### 2.2 Testing of Techniques

In order to test the various techniques, a large-scale virtual environment was simulated. In the paper, *Analysis of Broad-Phase Spatial Partitioning Optimisations in Collision Detection*, a similar simulation type is described (Glass). An axis aligned square with dimensions 5000 by 5000 was created. In this box, 1000 circles were spawned at random locations with constant speeds and random directions. Since narrow-phase collision is not the focus of this test, the use of circles was best as collision testing would be very simple. If the distance between the centers of the circles is less than the diameter, they collide. A very simple collision response was implemented. If the circles hit the outside bounds, the appropriate component of their velocity vector’s sign was switched. If two circles hit each other, they reversed in direction. In order to insure that the collision detection was occurring effectively, a simple display was constructed.

Based on this simulation two experiments were carried out. The amount of objects was increased by one every 100 frames until 2000 objects were in the virtual environment. In a separate experiment, the speed of each object was increased from one unit frame\(^{-1}\) by one every 100 frames until a speed of 50 unit frame\(^{-1}\) was reached. In both cases, each of the three techniques was used to handle the broad-phase collision detection. Each scenario was run for 100 frames in order to account for external fluctuations in CPU activity. When possible, the initial random object locations were stored and reused in order to provide some consistency. After each 100 frame segment, two metrics were recorded; the total runtime of collision detection (by using system time) and the amount of potential collisions reported (using a simple counter).

### 3 Theoretical Results

#### 3.1 Object Count

Based on the amount of steps that have to be carried out, it can be hypothesized that increasing the amount of objects would most significantly impact spatial hashing. This is because in each frame of the simulation, the spatial hashing algorithm would have to rehash all the objects, thus increasing the amount of tasks proportionately. On the other hand, the steps in sweep and prune and the hybrid method would not necessarily increase the amount of steps proportionately because the objects are stored in the relevant data structure only once. Due to the
more theoretically optimized nature of the hybrid method (less objects overlapping on one axis),
it can be predicted that it will consistently perform at a lower runtime.

3.2 Object Speed

Predicting the efficiency of algorithms with increasing object speed is based on their
reliance on temporal coherence. Since spatial hashing as a method does not rely on temporal
coherence, increasing the speed of the objects should not affect the average run time of the
collision detection significantly. On the other hand, increasing the speed of the objects will
increase the amount of swaps in each frame of sweep and prune leading to a significantly larger
amount of tasks and therefore a significantly smaller efficiency. The migrations of objects
between cells in the hybrid method should help reduce the amount of swaps that have to be
performed and thus should make the hybrid relatively more efficient.

4 Results and Analysis

The amount of potential collisions recorded was more of a proxy for efficiency because the
efficiency of the algorithms was not solely reliant on the amount of collisions reported, but
also on the amount of tasks needed to report the potential collisions. The runtime of collision
detection was therefore used more extensively in data analysis. Since, the total runtime was
calculated after each 100 frame interval, the amount was divided by 100 to give the average
runtime in ms frame⁻¹.

4.1 Increasing Object Amount

Due to the vast amount of data, the following three graphs only show every 20th data
point, when the object amount was increased by 20.

In the above graph it can be seen that increasing the amount of objects increases the run
time of collision detection for sweep and prune. At 1000 objects it took 4.06 ms frame⁻¹ to
perform collision detection. At 2000 objects it took 8.27 sec frame$^{-1}$ to carry out collision
detection. The average rate of increase was therefore 0.00421 ms frame$^{-1}$ object$^{-1}$.

Figure 4: Average runtime versus object count for Spatial
Hashing

The results for spatial hashing show that a significantly larger amount of time was needed
for collision detection and a steeper upwards trend was observed. At 1000 objects, the average
runtime was 14.82 ms frame$^{-1}$. It increased to 29.64 ms frame$^{-1}$ at 2000 objects, showing an
average increase rate of 0.01482 ms frame$^{-1}$ object$^{-1}$.

Figure 5: Average runtime versus object count for the Hybrid
method

The runtime for the hybrid method was more comparable to that of sweep and prune. At
1000 objects an average runtime of 4.99 ms frame$^{-1}$ was recorded. This increased to 6.71 ms
frame$^{-1}$ at 2000 objects. An average increase of 0.00172 ms frame$^{-1}$ object$^{-1}$ was observed. The trend of the hybrid method however, showed the most significant fluctuations.

The reason why spatial hashing was the least efficient of the methods and the fastest to deteriorate at larger scales was most likely its lack of use of temporal coherence. If the number of objects is denoted as $n$, in the best case for sweep and prune, and the hybrid method (when no objects swap positions) they will have to perform $n$ fewer sets of tasks than spatial hashing when updating positions. Although the best scenario is not always the case, they will almost always perform fewer tasks when updating positions than spatial hashing. With regards to the steep upwards trend, as the object amount is increased the amount of tasks per frame should increase almost proportionately for spatial hashing. The average rate of increase (0.01482 ms frame$^{-1}$ object$^{-1}$) suggested that this exactly happened as the runtime per object at 1000 objects was also 0.01482 ms frame$^{-1}$ object$^{-1}$. For sweep and prune and spatial hashing, increasing the number of objects in the best case scenario would only require the initial insertion of the object into the data structures and thus will not lead to a significant increase of runtime.

Contradicting the anticipated results, sweep and prune initially performed better than the hybrid method. The most likely reason why this occurred is the handling of cell migrations by the hybrid method. The hybrid method would be more efficient than sweep and prune in cases where there would be greater object density because the hybrid method is better at handling cases of objects overlapping on only one axis. At a lower object amount this is less likely to be the case. However, as more objects are added, the density is likely to increase at a greater rate than the required cell migrations. This is because there is limited area over which objects can cross cell bounds (they have to be close to the borders and have sufficient velocity). This explains why the sweep and prune deteriorates at a faster rate (0.00421 ms frame$^{-1}$ object$^{-1}$) than the hybrid method (0.00172 ms frame$^{-1}$ object$^{-1}$). This also explains the fluctuations in the trend for the hybrid method. The density of the randomized scenario may end up at varying degrees and require varying degrees of cell migration.

4.2 Increasing Object Speed

![Figure 6: Average runtime versus object speed for Sweep and Prune](image-url)
The average runtime that collision detection takes through sweep and prune increases with object speed. Initially at a speed of 1 unit frame\(^{-1}\) the average run time was 3.43 ms frame\(^{-1}\). It increased to 20.44 ms frame\(^{-1}\) at 50 unit frame\(^{-1}\). This is an average increase of 0.340 ms unit\(^{-1}\).

As seen in figure 7, there was very little variation in the runtime with increasing object speed. Throughout the experiment, the average run time fluctuated slightly around the average 13.16 ms frame\(^{-1}\). The average runtime at 1 unit frame\(^{-1}\) was 15.44 and was 13.42 ms frame\(^{-1}\) at 50 unit frame\(^{-1}\).
Once again, the trend with the hybrid method is similar to that of sweep and prune. At the initial speed, the average runtime was 4.06 ms frame\(^{-1}\). At 50 unit frame\(^{-1}\) the average runtime increased to 18.41 ms frame\(^{-1}\). Thus, the average increase was 0.287 ms unit\(^{-1}\).

Although initially the sweep and prune performed with the least average runtime, the efficiency of the algorithm deteriorated the fastest. Both sweep and prune and the hybrid method intersect with spatial hashing at around 30 unit frame\(^{-1}\). As hypothesized, the efficiency of spatial hashing did not deteriorate because of its independence from the movement of the objects. At every frame, the objects are rehashed into the right cells. On the other hand, since sweep and prune and the hybrid method rely on temporal coherence, greater object movement will increase the amount of swaps in their lists and will consequently increase the amount of tasks that have to be processed. Eventually with increasing object speed, spatial hashing will become the most efficient method.

Despite the trend being similar for sweep and prune and the hybrid, the hybrid method did not deteriorate as fast as sweep and prune. This is because there is a smaller limit to the amount the objects can move within the partitioned space. If they move beyond the bounds of a cell, the cell migration algorithm will run. The object will then be inserted into a position relatively close to where it should be in the lists instead of being swapped through the insertion sort. In the case of sweep and prune, if an object moves a large distance, it will have to be swapped many times through the list. With each of these swaps, a test for AABB overlap will be carried out and so runtime will significantly increase.

5 Conclusion

Based on the data collected and the trends observed, it can be concluded that the hybrid method would be an effective improvement on its individual components in certain scenarios. In terms of object amount, it will almost always be more efficient than the typical implementation of spatial hashing. In scenarios where the density of the objects in the space is not large however, sweep and prune may be more efficient. This is because its advantage over sweep and prune comes when there is a large amount of objects that only overlap on one axis. This is more frequent at high densities. In terms of object speed, after a certain point, spatial hashing would be more efficient than the hybrid method. However, before that point the data shows that the hybrid method will generally be more efficient than sweep and prune. Based on this, it can be concluded that the hybrid method is favourable over sweep and prune and spatial hashing at high objects amounts and at moderate object speeds.

5.1 Evaluation

Although the simulation provided a good idea about the capabilities of the hybrid method in relation to its two individual components, there were certain factors that could have led to inaccuracy or could have been lacking from the data. One of these factors was the randomized locations and directions of the circles. There was an attempt to control this in the speed investigation. The initial circle locations were stored and they were replicated with increasing speed. With the object count however, this was not done because the stored scenario would have had to be updated at every object addition. When the experiment moved to a different broad-phase method, the original scenario with only 1000 objects would have had to remain and the
subsequent locations of additional objects would have had to be stored. There was however the potential of having a text file with the initial locations of all 2000 objects that could have been read by the simulation.

There could have also been an improvement with the metrics collected to test for efficiency. The use of system time is subject to problems with lagging in the system, so it was not necessarily accurate. As well, memory allocation data was not collected in this experiment even though it was accounted for in the implementation of sweep and prune. Using a debugger to monitor the steps of the algorithm may have provided the most accurate way of measuring efficiency.

5.2 Future Work

More data could help with the data analysis. The point at which the hybrid method surpassed sweep in prune in efficiency was close to the final data point in both the experiments. Therefore, more runs of the simulation with even greater object amounts and speeds would significantly contribute to the analysis. Replicating the experiment with (unique) combinations of high object speeds and high object amounts could provide further insight about the scenarios in which the hybrid method is most appropriate.

Future testing of the hybrid method should also be carried out as there are more broad-phase collision detection methods to which it could be compared. There could be more efficient methods in the prescribed scenarios than the hybrid. As well, there are different alterations and optimizations that could be attempted to make the method more efficient. Spatial hashing is just one of many spatial partitioning techniques that can be used in the broad-phase of collision detection. Attempting sweep and prune in combination with one of the other methods (such as division by a hierarchal data tree) could yield better results. There could also be more accurate and efficient ways to handle cell migration than that which was used in this implementation. Cell migration was a pivotal part of the functionality of the hybrid method and would thus have a major impact. This further research will in general help improve the hybrid method in different large-scale scenarios and increase its viability for application in virtual simulations.

Word Count: Approximately 3879
6 Words Cited.


SweepAndPrune.java
import java.util.LinkedHashSet;
/**
 * A class that contains all the data structures
 * and methods necessary for sweep and prune.
 */
public class SweepAndPrune
{
    // instance fields
    protected Point[] xAxis = new Point[BroadPhaseSimulation.numberCircles*2];
    protected Point[] yAxis = new Point[BroadPhaseSimulation.numberCircles*2];
    protected LinkedHashSet<Pair> overlaps;
    private Circle[] circles = BroadPhaseSimulation.circles;

    /**
     * Constructs an object that handles sweep and prune.
     */
    public SweepAndPrune()
    {
        for (int i=0; i<BroadPhaseSimulation.numberCircles; i++)
        {
            xAxis[i*2] = circles[i].getMinX();
            xAxis[i*2+1] = circles[i].getMaxX();
            yAxis[i*2] = circles[i].getMinY();
            yAxis[i*2+1] = circles[i].getMaxY();
        } // end of for (int i=0; i<BroadPhaseSimulation.numberCircles; i++)
        overlaps = new LinkedHashSet<Pair>();
    } // end of constructor SweepAndPrune()

    /**
     * Sorts the x and y axis through an insertion sort while
     * recording possible collision pairs.
     */
    public LinkedHashSet<Pair> sort()
    {
        // insertion sort on x-axis
        Point temp;
        int j;
        for (int i=1; i<xAxis.length; i++)
        {
            temp=xAxis[i];
            for (j=i-1; j>=0; j--)
            {
                if (xAxis[j].getValue()>=temp.getValue())
                {
                    xAxis[j+1]=xAxis[j];
                    checkOverlap(temp.getOwner().getIndex(),
                    xAxis[j].getOwner().getIndex());
                }
                else
                {
                    break;
                } // end of if (xAxis[j].getValue()>=temp.getValue())
            } // end of for (j=i-1; j>=0; j--)
        } // end of for (int i=1; i<xAxis.length; i++)
    }
}
} // end of for (j=i-1; j>=0; j--)
xAxis[j+1]=temp;
} // end of for (int i=1; i<xAxis.length; i++)

// insertion sort on y-axis
Point temp2;
for (int i=1; i<yAxis.length; i++)
{
temp2=yAxis[i];
for (j=i-1; j>=0; j--)
{
if (yAxis[j].getValue()>=temp2.getValue())
{
    yAxis[j+1]=yAxis[j];
    checkOverlap (temp2.getOwner().getIndex(),
yAxis[j].getOwner().getIndex());
}
else
{
    break;
}
} // end of for (j=i-1; j>=0; j--)
yAxis[j+1]=temp2;
} // end of for (int i=1; i<yAxis.length; i++)
return overlaps;
} // end of method LinkedHashSet<Pair> sort()

/**
 * Checks for overlap between two circles.
 */
private void checkOverlap(int one, int two)
{
    Pair pair=new Pair (one, two);
    if (Math.abs(circles[one].getMinX().getValue() - circles[two].getMinX().getValue())
    <=BroadPhaseSimulation.RADIUS*2 &&
    Math.abs(circles[one].getMinY().getValue() -
circles[two].getMinY().getValue())
    <=BroadPhaseSimulation.RADIUS*2)
    {
        overlaps.add(pair);
    }
else
{
    overlaps.remove(pair);
} // end of if (Math.abs(circles[one].getMinX().getValue() ...
} // end of method checkOverlap(int one, int two)

SpatialHash.java
import java.util.ArrayList;
import java.util.Hashtable;

/**
 * A class that contains all the data and methods for
 * spatial hashing.
 */
public class SpatialHash {
    // instance fields
    private int maxx;
    private int maxy;
    private int size;
    // the dimensions of the screen and cells
    Hashtable<Integer, ArrayList<Integer>> index;

    /**
     * Constructs a object to handle spatial hashing.
     */
    public SpatialHash(int x, int y, int size) {
        maxx = x;
        maxy = y;
        this.size = size;
        index = new Hashtable<Integer, ArrayList<Integer>>(maxx/size*maxy/size);
    } // end of constructor SpatialHash (int x, int y, int size)

    /**
     * Clears the hash table contained by this object.
     */
    public void clear() {
        index.clear();
    } // end of method clear()

    /**
     * Finds all the cells into which the circle falls.
     */
    private ArrayList<Integer> getLocations(Circle circle) {
        ArrayList<Integer> locations = new ArrayList<Integer>();
        Integer location = (int) (Math.floor(circle.getMinX().getValue()/size) +
                (Math.floor(circle.getMinY().getValue()/size)-1)*maxx/size);
        locations.add(location);
        location = (int) (Math.floor(circle.getMaxX().getValue()/size) +
                (Math.floor(circle.getMinY().getValue()/size)-1)*maxx/size);
        if (!locations.contains(location))
            locations.add(location);
        location = (int) (Math.floor(circle.getMinX().getValue()/size) +
                (Math.floor(circle.getMaxY().getValue()/size)-1)*maxx/size);
        if (!locations.contains(location))
            locations.add(location);
        location = (int) (Math.floor(circle.getMaxX().getValue()/size) +
                (Math.floor(circle.getMaxY().getValue()/size)-1)*maxx/size);
        if (!locations.contains(location))
            locations.add(location);
        return locations;
    } // end of method getLocations(Circle circle)

    /**
     * Registres the circle in the correct cells.
     */
    public void register(Circle circle)
    {
{ ArrayList<Integer> locations = getLocations(circle);

for (int i=0; i<locations.size(); i++)
{
    ArrayList<Integer> destination;
    destination = index.get(locations.get(i));
    if (destination==null)
    {
        destination=new ArrayList<Integer>();
    } // end of if(destination==null)
    destination.add(circle.getIndex());
    index.put(locations.get(i), destination);
} // end of for (int i=0; i<locations.size(); i++)

} // end of method register

/**
 * Reports the circles close to the specified one.
 */
public ArrayList<Integer> report(Circle circle)
{
    ArrayList<Integer> near=new ArrayList<Integer>();
    ArrayList<Integer> locations=getLocations(circle);
    for (int i=0; i < locations.size(); i++)
    {
        if (index.get(locations.get(i))!=null)
        {
            index.get(locations.get(i));
            near.addAll(index.get(locations.get(i)));
            near.remove(new Integer(circle.getIndex()));
        } // end of if (index.get(locations.get(i))!=null)
    } // end of for (int i=0; i< locations.size(); i++)
    return near;
} // end of method report(Circle circle)

} // end of class SpatialHashing

SweepAndPruneH.java:
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedHashSet;

/**
 * A class to handle the sweep and prune in
 * the cells of the spatial hash.
 */
public class SweepAndPruneH extends SweepAndPrune
{
    ArrayList<Point> xAxis2;
    ArrayList<Point> yAxis2;
    int minx;
    int miny;
    int maxx;
    int maxy;
/**
 * Constructs an object to handle sweep and prune
 * in the cells of a spatial hash.
 */

public SweepAndPruneH (int minx, int miny, int maxx, int maxy)
{
    overlaps= new LinkedHashSet<Pair>();
    xAxis2=new ArrayList<Point>();
    yAxis2=new ArrayList<Point>();
    this.minx=minx;
    this.miny=miny;
    this.maxx=maxx;
    this.maxy=maxy;
} // end of constructor SweepAndPruneH (int minx, int miny, int maxx, int maxy)

/**
 * Sorts a specific cell along the x and y axis
 * and returns potential overlaps.
 */

public LinkedHashSet<Pair> sort()
{
    xAxis=new Point[xAxis2.size()];
    xAxis2.toArray(xAxis);
    yAxis=new Point[yAxis2.size()];
    yAxis2.toArray(yAxis);
    super.sort();
    xAxis2=new ArrayList<Point>(Arrays.asList(xAxis));
    yAxis2=new ArrayList<Point>(Arrays.asList(yAxis));
    return overlaps;
} // end of method sort()

/**
 * Updates the circles in each cell accounting
 * for circle "migration."
 */

public void update()
{
    if (minx!=0)
    {
        for (int i=0; i<xAxis.length; i++)
        {
            // removes vertices who have left the area (less than the minimum x)
            if (xAxis[i].getValue() < minx)      
            
                xAxis2.remove(xAxis[i]);
                xAxis2.remove(xAxis[i].getOwner().getMinX());
                xAxis2.remove(xAxis[i].getOwner().getMaxX());
                yAxis2.remove(xAxis[i].getOwner().getMaxY());
                yAxis2.remove(xAxis[i].getOwner().getMinY());
                BroadPhaseSimulation.hybrid.register(xAxis[i].getOwner());
            
            else
            
                break;
        } // end of if (xAxis[i].getValue() < minx)
    } // end of for (int i=0; i<xAxis.length; i++)
if (maxx!=5000)
{
    for (int i=xAxis.length-1; i>=0; i--)
    {
        // removes vertices who have left the area (greater than the maximum x)
        if (xAxis[i].getValue()>maxx)
        {
            xAxis2.remove(xAxis[i]);
            xAxis2.remove(xAxis[i].getOwner().getMaxX());
            xAxis2.remove(xAxis[i].getOwner().getMinX());
            yAxis2.remove(xAxis[i].getOwner().getMaxY());
            yAxis2.remove(xAxis[i].getOwner().getMinY());
            BroadPhaseSimulation.hybrid.register(xAxis[i].getOwner());
        }
        else
        {
            break;
        }
    }
    // end of for (int i=xAxis.length-1; i>=0; i--)
}
// end of if (maxx!=5000)

if (miny!=0)
{
    for (int i=0; i<yAxis.length; i++)
    {
        // removes vertices who have left the area (less than the minimum x)
        if (yAxis[i].getValue()<miny)
        {
            yAxis2.remove(yAxis[i]);
            xAxis2.remove(yAxis[i].getOwner().getMinX());
            xAxis2.remove(yAxis[i].getOwner().getMaxX());
            yAxis2.remove(yAxis[i].getOwner().getMaxY());
            yAxis2.remove(yAxis[i].getOwner().getMinY());
            BroadPhaseSimulation.hybrid.register(yAxis[i].getOwner());
        }
        else
        {
            break;
        } // end of if (yAxis[i].getValue()<miny)
    }
    // end of for (int i=0; i<yAxis.length; i++)
}
// end of if (miny!=0)

if (maxy!=5000)
{
    for (int i=yAxis.length-1; i>=0; i--)
    {
        // removes vertices who have left the area (greater than the maximum x)
        if (yAxis[i].getValue()>maxy)
        {
            yAxis2.remove(yAxis[i]);
            xAxis2.remove(yAxis[i].getOwner().getMaxX());
            xAxis2.remove(yAxis[i].getOwner().getMinX());
            yAxis2.remove(yAxis[i].getOwner().getMaxY());
            yAxis2.remove(yAxis[i].getOwner().getMinY());
        }
    }
    // end of if (maxy!=5000)
public class SpatialHashH {
    private int maxx;
    private int maxy;
    private int size;
    private Hashtable<Integer, SweepAndPruneH> index;

    public SpatialHashH(int x, int y, int cellSize) {
        maxx = x;
        maxy = y;
        size = cellSize;
        index = new Hashtable<Integer, SweepAndPruneH>(maxx/size*maxy/size);
    }

    public void register(Circle circle) {
        registerPoint(circle.getMinX(), circle.getMinY());
        registerPoint(circle.getMaxX(), circle.getMinY());
        registerPoint(circle.getMinX(), circle.getMaxY());
        registerPoint(circle.getMaxX(), circle.getMaxY());
    }

    private void registerPoint(Point x, Point y) {
        Integer location = (int)(Math.floor(x.getValue()/size) +
                                 (Math.floor(y.getValue()/size))*maxx/size);
        if (index.get(location) == null) {
            location = (int)(Math.floor(x.getValue()/size) +
                             (Math.floor(y.getValue()/size))*maxx/size);
        }
    }
}
index.put(location, new SweepAndPruneH
((location%((maxx/size)*size)).location/(maxx/size)*size,
(location%((maxx/size)+1)*size),(location/(maxx/size)+1)*size));
} // end of if (index.get(location)==null)
if (!index.get(location).xAxis2.contains(x)) {
  if (Math.abs(x.getValue()-index.get(location).minx)<Math.abs(x.getValue()­
  index.get(location).maxx))
    index.get (location).xAxis2.add(0, x);
  else
    index.get (location).xAxis2.add(x);
} // end of if (!index.get(location).xAxis2.contains(x))
if (!index.get(location).yAxis2.contains(y)) {
  if (Math.abs(y.getValue()-index.get(location).miny)<Math.abs(y.getValue()­
  index.get(location).maxy))
    index.get (location).yAxis2.add(0, y);
  else
    index.get (location).yAxis2.add(y);
} // end of if ( ! index.get(location).yAxis2.contains(y))
} // end of method registerPoint()

/**
 * Updates the positions in each cell.
 */
public void update()
{
  for (int i=0; i<maxx/size*maxy/size; i++)
  {
    if (index.get(i)!=null)
    {
      index.get(i).update();
    } // end of if (index.get(i)!=null)
  } // end of for (int i=0; i<maxx/size*maxy/size; i++)
} // end of method update()

/**
 * Reports the pairs of potential collisions in each cell.
 */
public void report()
{
  for (int i=0; i<maxx/size*maxy/size; i++)
  {
    if (index.get(i)!=null)
    {
      BroadPhaseSimulation.check(index.get(i).sort());
    } // end of if (index.get(i)!=null)
  } // end of for (int i=0; i<maxx/size*maxy/size; i++)
} // end of method report()
} // end of class SpatialHashH