COMP 768 - Physically Based Simulation Homework 2 - Collision Detection Between Rigid Bodies

Abhinav Golas

March 17, 2009

1 Part A - Assumptions and Algorithm

It is given that the objects are sphere like, i.e. the variance of points is similar along each and every axis. This makes it clear that cubical or spherical bounding boxes would be ideal choices. Also, it is clear that the bounding box would not need to be refitted at every step, since even under rotation, the existing bounding box would suffice due to the objects having the same aspect ratio in all axes. We can use a 2 step algorithm for doing collision testing. For this we maintain 2 copies of the scene, one containing full resolution objects, with their bounding boxes (S_1), and the other containing simple axis-aligned cubical bounding boxes (S_2). The reason for this partition is to take advantage of the fact that bounding boxes need not be updated. To utilize this advantage, the scene S_2 is not given rotation matrix updates, only translation updates. Also, S_1 is only updated whenever there is a collision detected in S_2, and even then, only those pairs are checked which show the possibility of a collision.

However, when we look at the relative expense of an axis aligned bounding box update versus the time taken for collision check stage, the difference is of nearly 2 orders of magnitude, which makes this algorithm not worth the effort, since we would only be reducing a very small portion of the time taken, while the bulk of the time taken would remain the same. A detailed basis for this is given in the analysis section.

I am using the SWIFT++ collision detection software as the base source.

2 Modifications for Part C

For part C, we are given the additional constraint on the scene that every object is of the same size. Considering the algorithm described in the previous section, there is an indication that we can improve the collision detection for scene S_2, since there is a basic unit of measurement for the scene, d_{max}, which is the extent of any object along any axis in the scene. That considered, the use of a

grid for speeding up adjacency queries presents itself. We can define a grid of size $n \times n \times n$ where $n = \frac{D}{d_{max}}$, D being the edge length of the cubical scene. Then, instead of tracking m objects, we need to track m points, each point representing the location of the centroid of the cubical bounding box of each object. Then, if object A lies in cell (i,j,k), then we only need to check any objects lying in the adjacent 27 cells for possible collisions. This is because that when the 2 objects intersect, the distance between their centroids must be less than d_{max}, which implies that points representing these objects can only lie in adjacent cells in this grid. Even in that case, the only check we need to do is to see whether the distance between the centroids is less than d_{max}. Also, for exact collision detection, we can avoid sorting all objects, instead sorting only the objects in consideration. The SWIFT toolkit offers an option of Local sort, which can be utilized for this step.

3 Analysis and Comparison

Now we present the analysis and comparison for the above algorithms.

3.1 Number of objects

First we analyze the growth of collision query time when the number of objects is increased. For this test we used objects of bounding box radius 1, mesh detail being a uniform distribution with mean 3, standard deviation 1. The important data in this test is the time per collision query. As expected, the collision time is the same irrespective of whether there are any collisions or not. This distinction has been denoted as positive and negative queries, positive queries being the ones in which collisions actually took place. The graph shows that the collision query time grows linearly with the number of objects (Figure 1). This conclusion can be tested with a loglog plot, i.e. a plot of log(x) vs. log(y). We

Table 1: Collision query behavior vs. number of objects

No. of objects	Time per collision query	No. of calls	Time per positive query	No. of positive calls	Update time
1	0.000167	1338	0.000142	19	0.000024
5	0.000734	1558	0.000713	151	0.000053
10	0.001611	1252	0.001734	243	0.000076
15	0.002131	1493	0.00202	836	0.000083
20	0.003234	2120	0.003306		0.000111
25	0.003894	1780	0.00382	748	0.000115
30	0.00501	2309	0.005103	1412	0.000126
35	0.005783	2121	0.00579	1027	0.000117
40	0.006812	1055	0.006774	643	0.000131
45	0.007558	2168	0.007552	1988	0.00014
50	0.008915	1823	0.008895	1525	0.000149

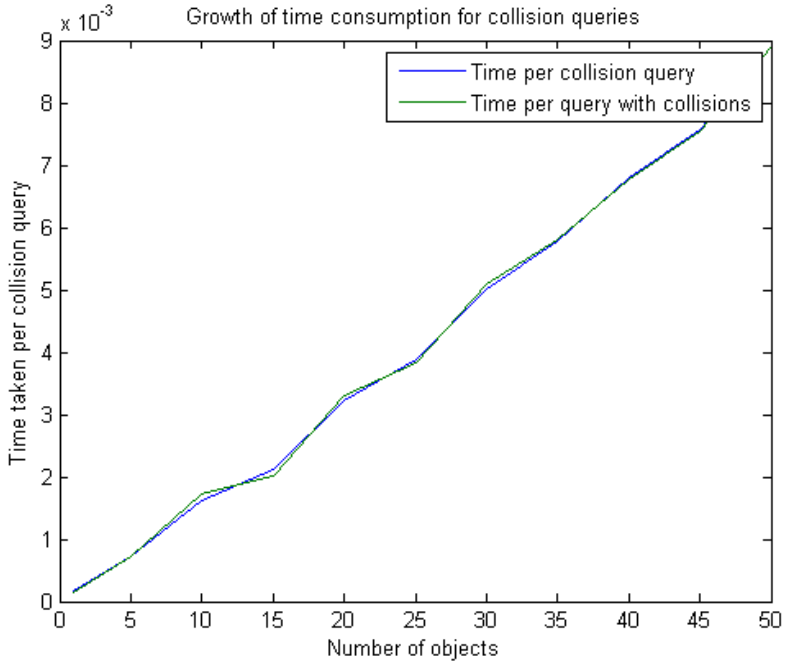

Figure 1: Growth of collision query time

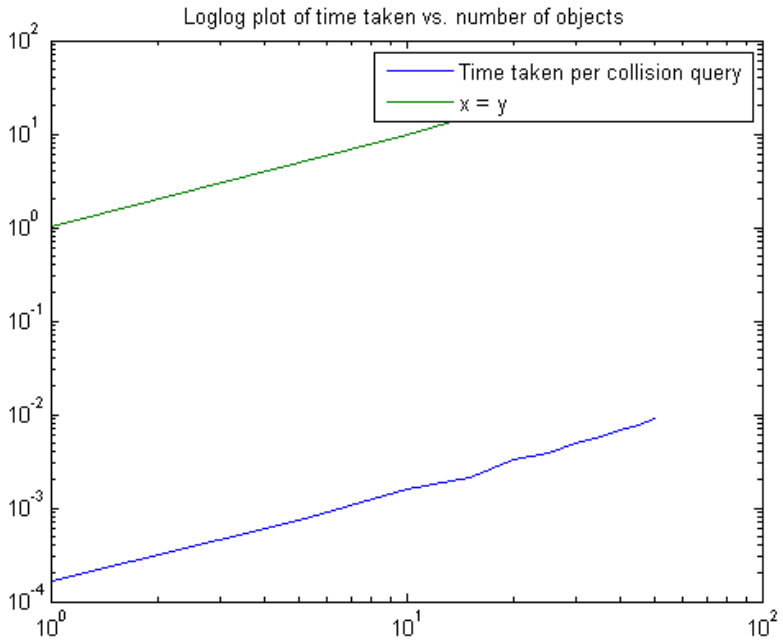

Figure 2: Loglog plot of collision query time

plot the loglog plot of x=y as a comparison. Since the slope of both curves is the same, we can conclude that the growth is linear (Figure 2). A miscellaneous observation which is expected, is that the percentage of positive queries in the collision queries grows linearly as well.

We now look at the possible reasons for this behavior. It can be expected that the sweep and prune step will reject many possible pairs. Since the aspect ratio of the objects is same along all axes, it is expected that the cubical bounding box utilized by SWIFT will fit the objects tightly, and thus reduce the chances of false positives, i.e. the BVH test for a pair of objects will be more likely to give the answer to the exact collision test. Also, a weaker extension of the tightness claim should also extend to the BVH hierarchy, which would result in performance improvements, and thus the near linear growth that we observe for the next part.

3.2 Complexity of objects

Table 2: Collision query behavior vs. complexity of objects

Mesh detail	Time per collision query	No. of calls	Time per positive query	No. of positive calls	Update time
2	0.002932	2681	0.002991	1096	0.0001
5	0.003699	2671	0.003608	886	0.000107
10	0.004375	2847	0.00453	679	0.000098
15	0.005002	2932	0.005025	2078	0.000085
20	0.005067	2060	0.005057	1317	0.000081
25	0.005475	1878	0.005428	252	0.000075
35	0.007121	1175	0.007502	568	0.00012

The graph shows that the growth of collision query time vs. polygon count is approximately linear, though the curve is not a perfect fit. It seems that the behavior is somewhere between linear and quadratic. This behavior is expected as the worst case performance of collision checking is $O(n^2)$, in which case we check every polygon with each other. To explain this, we can extend from the argument from the previous section. The tightness of the bounding boxes due to aspect ratios being similar along all axes reduce the number of object pairs that we need to test. Hence, even though we may use $O(n^2)$ overlap tests for an object pair, these set of tests will not need to be applied for all pairs of objects. So, on an average, the performance will not be quadratic, but near linear, since the number of object pairs for which we run the overlap tests is reduced.

3.3 Size of objects

With the increase in object radius, the bounding cube in which the scene is contained, is bound to get more packed. As this happens, the bounding boxes

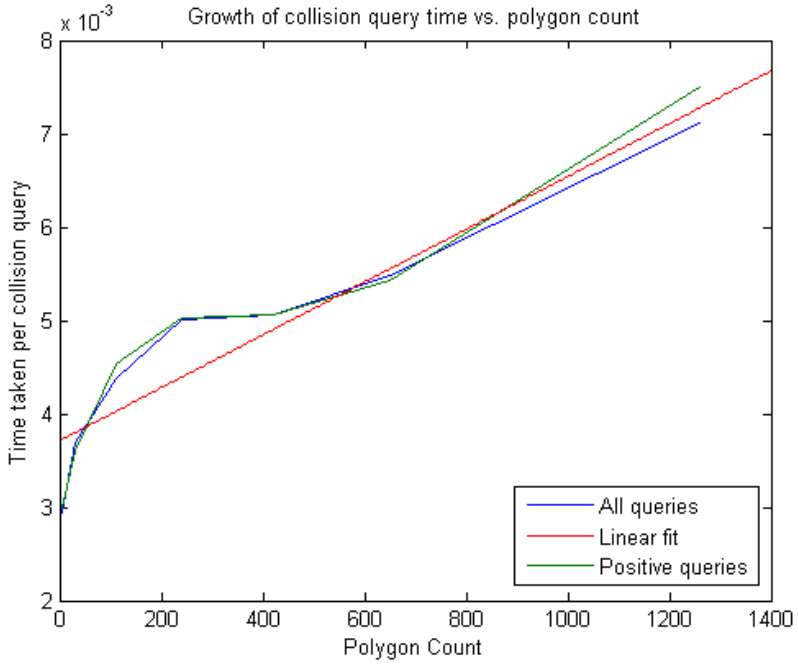

Figure 3: Growth of collision query time with polygon count

Table 3: Collision query behavior vs. size of objects

Radius	Time per collision query	No. of calls	Time per positive query	No. of positive calls	Update time
0.5	0.003012	1485	0.002831	309	0.0001
1	0.003164	989	0.003171	250	0.000099
2	0.00319	710	0.00319	710	0.00009
3	0.003215	659	0.003215	659	0.000095
5	0.003324	979	0.003324	979	0.0001

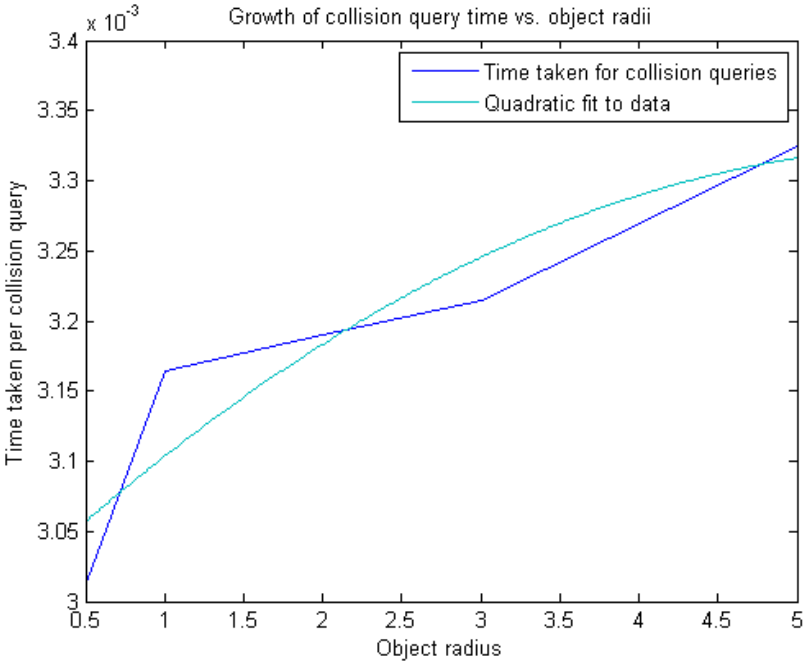

Figure 4: Growth of collision query time with object radii

of each object in the scene are more likely to intersect, and so this pre-processing step will quickly become less and less useful. But as the scene moves towards saturation, i.e. perfect packing, the increase in time will start to plateau as the exact collision detection step will start to dominate, and the sweep and prune step will give almost the entire object set as possible collision pairs. This is also seen in the graph, and the curve fit reflects that the data is expected to plateau.

3.4 Comparison of new algorithm in Part C with Part A algorithm

Table 4: Comparison of both algorithms

	Time per collision query	No. of calls	Time per positive query	No. of positive calls	Update time
Part A	0.003164	989	0.003171	250	0.000099
Part C	0.000527	291	0.000525	147	0.065

As we can see from the above table, the modifications proposed, result in drastic improvements in the collision query times. However, there is a corresponding increase in the time taken to update the scene. The problem can be traced back to the fact that if the radius is small, the grid we constructed for speeding up neighbor queries will be very large, e.g. 40^3 for our given case. At each step we will need to clear and update this grid, which results in the increase in time consumed. We can avoid this by using a more efficient representation for the grid, storing only those cells which contain objects. I experimented with a hash based storage for the grid, but could not make a functional prototype in time. That would reduce the clearing loop from 40^3 to n, where n is the number of objects. The update step is linear in number of objects and hence is not a bottleneck.

The fixed size condition is also utilized in a pre-process step, we check whether the centers of mass of the 2 objects under consideration are more than d_{max} apart, in which case they cannot intersect. This is analogous to having a spherical bounding box around the object. Using a pruned neighborhood using the grid, and the simple distance based pruning, the performance of the collision query is improved. If the update performance can be improved by the modifications recommended above, we can achieve much better performance.

In conclusion, we can achieve good performance for collision checking on large numbers of objects, if the objects have similar aspect ratios and sizes.