An Approach to Line-Circle Collision Detection and Response

Eric Leong

William A. Shine Great Neck South High School

November 18, 2009

Introduction

Software simulations are being utilized in more and more fields to model the behavior of physical systems. Physical simulations are seen in virtual reality, video gaming, and computer-assisted drawing tools (Mirtich B. , 2000). Simulators attempt to calculate the outcomes of interactions between physical entities: one of these interactions is collision between bodies. Collision simulation has two aspects, collision detection and response, which are separate in theory but intertwined in practice, as how one is implemented affects the other (Moore & Williams, 1988).

A physics simulation attempts to model the interactions between objects by breaking down the movement of objects into discrete intervals (Mirtich B. V., 1996). This way, frames, which are snapshots of the simulation at an instantaneous point in time, can be drawn and then displayed onto the screen. If these frames are displayed fast enough, the perception of smoothness can be achieved, resulting in a real-time simulation that responds nearly instantly to user interactions (Cohen, Ming, Manocha, & Ponamgi, 1995; Hubbard, 1995; O'Sullivan & Dingliana, 2001). Discrete positions and velocities are used rather than continuous parametric functions to describe the movement of objects in order to simplify complex movements, so an object moves through translation of the object in the direction and magnitude of the vector, as seen in Figure 1 (Mirtich B. V., 1996).

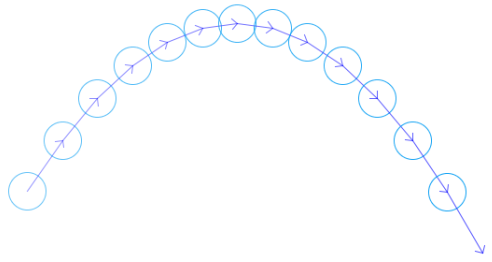


Figure 1. The movement of the circle is split up into discrete positions and velocities at certain times (increasing opacity indicates the passage of time), approximating a continuous parametric function, a parabola in this case.

The objects in a physics simulation can simply be stored as a list of objects with positions and velocities. Detecting a collision consists of determining whether objects will intersect or have intersected, while responding to a collision involves calculating the new positions and velocities for each object involved (Hubbard, 1995).

Collision Detection

A collision occurs when two objects intersect or are about to intersect, meaning that they are touching or travelling toward each other (Moore & Williams, 1988). A common method of collision detection, called *pair-processing*, starts by comparing an object with another object to see if a collision has occurred between those two objects, so at any one moment in time, a pair of objects is being compared. If the two objects, lines and circles, are separated into separate lists, one of size n and the other of size m, and each circle is only compared to every line or vice versa, then the worst-case, average, and best-case running time of the algorithm in big-O notation is $O(nm)$, which makes it a linear relationship between number of one type of object and running time.

The collision detection process can be done two different ways: detecting a collision before it occurs or after. The two approaches are mainly differentiated by speed and accuracy (Redon, Kheddar, & Coquillart, 2002). Speed is measured in terms of frames per second, which is the inverse of the time needed to calculate one frame. If an object is to move a certain amount, d, over a certain time, t, then between the two frames the object is moved a distance d, but the time elapsed between two frames should be t seconds. As long as the amount of time it takes to calculate one frame is less than t, the processor can be idle after the calculation is completed until t seconds has passed. Accuracy is more abstract: a minor inaccuracy would be an approximate location, while not detecting a collision would

be a major error. It is faster, but less accurate to do collision detection after the collision has already occurred in the simulation than it is to predict the location of a collision(Redon, Kheddar, & Coquillart, 2002).

For detection after a collision, also known as retroactive detection, objects are moved forward in small, finite amounts, simultaneously frame-by-frame until an intersection or overlapping is detected between objects (see Figure 2)(Mirtich B. , 2000). This may be interpreted as "stepping" the objects forward by a small amount, running through the list of objects and checking each pair for collisions. The collision is then resolved either by applying a de-penetration force or stepping back the objects until they no longer collide and the process begins again (Mirtich B. V., 1996; Mirtich B. , 2000). In psuedocode, this can be written as:

```
main_loop(){
      move_objects_forward();
      test_for_collisions();
}
```

The main benefit of this method is speed, since checking for collisions between two static objects is fairly fast – for example, collision checking between two circles only involves the distance formula (Hubbard, 1996; Heuvel & Jackson, 2002). If the objects are moved by a distance greater than their size, "tunneling" through other objects that they should have collided with may occur, since the objects do not intersect in any of the frames (see Figure 2)(Redon, Kheddar, & Coquillart, 2002). The chance of this occurring can be reduced by decreasing the time interval between frames so that the distance travelled between successive checks is lower, but the problem will still be there, as it is always possible to make an object small enough to fit between the distance travelled (Hubbard, 1996; Hahn, 1988). Also, the speed of the algorithm will be reduced due to the additional

checks for intersection, to the point where it is unusable (Moore & Williams, 1988; Mirtich B. , 2000).

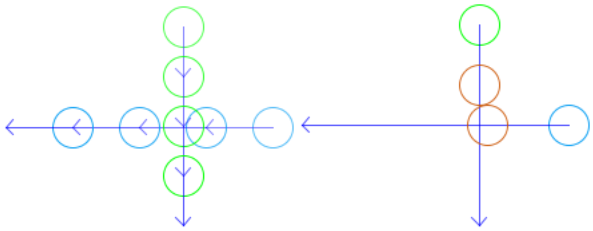


Figure 2. Diagrams generated by the collision detection algorithm discussed later in this paper, showing the results of two different collision methods. On the left, the objects are moved forward in small increments and intersections are only done between the final positions, resulting in tunneling. On the right, the collision is detected (in orange) when collision detection is done before the collision occurs

Detecting a collision before it occurs is less likely to miss a collision, as the objects do not intersect at the time of the collision (Mirtich B. V., 1996). As an object moves through time, it sweeps a path known as a space-time bounding structure. If this path overlaps with the path swept by another object, a collision is likely to occur (see Figure 3)(Hubbard, 1995). The intersection of two such areas can be solved relatively quickly using matrices and Cramer's rule (Canale & Chapra, 2001). If there is overlapping, the exact location of the collision can be solved for using parametric equations(Heuvel & Jackson, 2002).

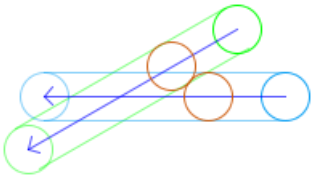


Figure 3. The green and blue circles sweep an area as they move along their movement vector. The overlapping of indicates that there is a possibility of a collision occurring between the two circles.

The end result is that for each object involved in a collision, the distance to the collision and a reference to the other object involved are stored. This is enough information for a collision response to be calculated for both objects.

Collision Response

After a collision is found, objects in a simulation are expected to *react* to it. Momentum and energy must be conserved after a collision, a rule followed by an *impulse-based* model for circles derived in *Gamasutra* (Heuvel & Jackson, 2002):

$$\vec{n} = \langle x_1, y_1 \rangle - \langle x_2, y_2 \rangle$$

$\vec{n}$ is the vector from the two points of collision for the two objects, $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$. The final velocity of the first object can be calculated using m, the mass of the objects; $\vec{v}_i$, the initial velocity of the objects, and $\vec{n}$, using this formula.

$$\vec{v}_{1_f} = \vec{v}_{1_i} - \vec{n} \cdot m_2 \frac{2\left(\vec{v}_{1_i} \cdot \vec{n} - \vec{v}_{2_i} \cdot \vec{n}\right)}{m_1 + m_2}$$

With that, the new velocity can be calculated using the new positions and the collision detection and response is complete. This simple model works reasonably well for simulating dynamic properties of objects, such as sliding and rolling (Mirtich B. V., 1996).

This model isn't very useful, as it assumes that energy is conserved, making the objects are elastic. But in reality, interactions result in energy lost, such as in the form of heat or sound, making the collision is inelastic, so the coefficient of restitution, C_R, is needed to determine the final velocities of both objects (Schwager & Pöschel, 1998). If it is 0, then a collision is perfectly elastic, if it is 1, then a collision is perfectly inelastic and all energy is lost (see Figure 4). The new final velocity, $\vec{v}_{f_2}$, is the original final velocity, $\vec{v}_{f_1}$, of the object multiplied by $1 - C_R$.

$$\vec{v}_{f_2} = \vec{v}_{f_1} \cdot (1 - C_R)$$

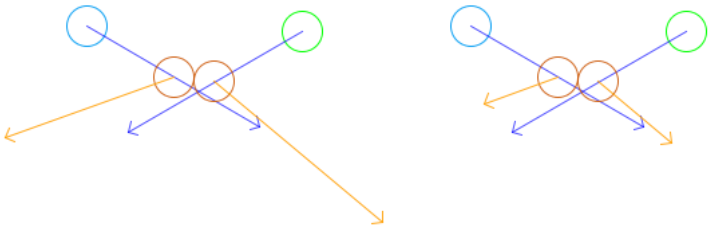


Figure 4. On the left is a perfectly elastic collision; on the right is an inelastic collision with a coefficient of restitution of .5. As a result, the resultant vector lines represented by the orange arrows are shorter in the collision on the right.

The collision detection and response model used in this paper builds upon this approach, handling multiple collisions during a single frame. This approach is detailed in a previous paper, "An Approach to Multiple Collision and Response," and results in smoother animations and interactions at a small cost in speed.

Circles and Lines

Collision between dynamic circles and static lines are found quite frequently in video games and animations, where lines may represent detailed immobile objects while circles represent approximations of less detailed, but moving objects. Since circles have an area and lines do not, any false positives or false negatives are quite obvious and undesirable, such as when a circle goes through a line. This is even more obvious when a line is not a part of a closed polygon and has exposed endpoints, since there are no other lines that could be visually interpreted as having been the "actual" collision by the user.

The focus of this paper is on single, independent static line segments, where both endpoints are exposed. These lines have zero thickness and a finite length, so traditional techniques that involve area or volume, such as the circle-circle collision detection mentioned earlier do not apply. Endpoints in this case cannot be treated as a line because they are not lines, yet must also have zero area. Circles on the other hand sweep an area as they travel, and if any part of that area intersects with the line, a collision has occurred.

Yet lines are simpler than other objects because they can be described by a single simple function, which simplifies the mathematics involved in collision detection. Zero thickness is a lurking issue here, since *x* and *y* values may not match up exactly with what is required due to floating point rounding, which occurs because the computer cannot store an infinite number of decimal places and must truncate or round after a certain point.

Goal

The line-circle collision detection and response algorithm will be built using Java 1.6 on top of a framework used in "An Approach to Multiple Collision Detection and Response." The inputs are the position, velocities, and mass of a moving circle and the position of a static line. This allows for a variety of situations to be placed before the algorithm that must be accounted for, even impossible situations involving the circle intersecting with the line. In turn, the algorithm is flexible enough to handle unusual situations.

After fulfilling those requirements, the algorithm should be able to run at an acceptable speed with a reasonable number of circles and lines – 100 lines with a length of 20 pixels and 100 circles with a radius of 10 pixels at 60 frames per second, the average refresh rate of a monitor. The refresh rate of a common LCD monitor is 60 Hz, which means that the image is refreshed 60 times per second. Rendering additional frames beyond that number is wasteful. This translates to 16.67 milliseconds for the algorithm to compute one frame. The unit of measurement for the size of the circles and lines does not matter, what matters is the relative size of the lines and circles. The actual size is only a visual choice.

Algorithm

The most important function used is the `closestpointonline` function. It takes as inputs the *x* and *y* coordinates of a point and the endpoints of a line and finds the closest

point on the line to the point. It is implemented using Kramer's rule, which returns the solution to a system of equations.

The method takes advantage of the fact that line with the shortest length that connects a line and a point must be perpendicular to the line. It starts off by generating the standard form of a line for the line, $ax + by = c$ by using the endpoints of the line. Then it creates another line in the standard form by flipping and negating the left coefficients of the first line to create a perpendicular line, then shifts the perpendicular line to contain the point that was passed as an argument. $-bp_x + ap_y = c$, where p is the point that was given. At this point, there is the given line and a perpendicular line that contains the given point. This is where Kramer's rule comes in and using the two equations, one can solve for the location of the closest point on the line.

Below is the implementation of the method described above with one small edge case added: if the given point is on the line, then the given point is closest.

```
public static final Point closestpointonline(float lx1, float ly1,
              float lx2, float ly2, float x0, float y0){
       float A1 = (ly2 - ly1);
       float B1 = (lx1 - lx2);
       double C1 = (ly2 - ly1)*lx1 + (lx1 - lx2)*ly1;
       double C2 = -B1*x0 + A1*y0;
       double det = (A1*A1 - (-B1*B1));
       double cx = 0;
       double cy = 0;
       if(det != 0){
              cx = (float)((A1*C1 - B1*C2)/det);
              cy = (float)((A1*C2 - -B1*C1)/det);
       }else{
              cx = x0;
              cy = y0;
       }
       return new Point(cx, cy);
}
```

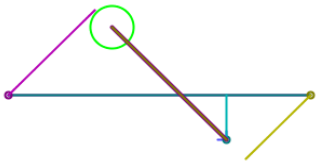


Figure 5. The bright lines connect the closest point on the line to the given point, which is a darker color. The dark colors represent the given point and lines. There are three cases where the algorithm is used, in magenta, blue, and gold.

Collision Detection

The collision detection algorithm starts by checking if a collision has occurred at all. Two types of collision might occur – the circle may be initially intersecting with the line or the path of the circle intersects with the line. If there is no intersection or interpenetration, then the algorithm can easily exit.

To find if the has initially intersected with the line, as seen in Figure 6, the closest point to line function can be used with the center of the circle as the point, and if the distance to the point returned by the function is smaller than the radius, there is a collision. But first, the point returned must also be checked to see if it is on the line segment, and this is done using the min and max functions. The point returned is definitely on the line, so if the *x* values are between the minimum and maximum *x* values of the endpoints of the line and the same is true for the *y* axis, then the point returned is on the line segment. Special consideration must be made for the endpoints. This is done by also checking if the endpoints are less than the radius away using the distance formula:

$d = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$.

If the path of the circle intersects with the line, as seen in Figure 6, then a similar method to the one employed above may be followed, except one simply just solves for the point of intersection between the movement vector and the line. Again, endpoints must be treated differently, and the final position of the circle must also be tested separately similarly to how the initial position circle was tested.

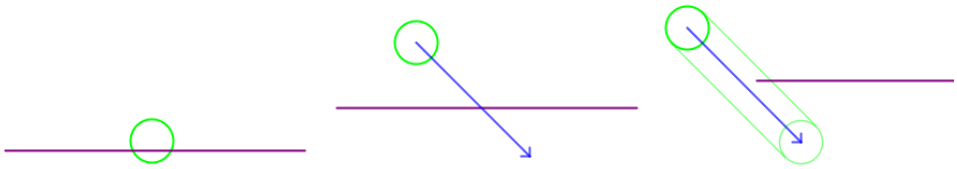


Figure 6. The green circle is intersecting with the line in the left diagram, while in the center diagram, the path of the circle intersects with the line. Lastly, in the right diagram, the area swept by the circle intersects with the line.

One also needs to check of the area swept out by the circle due to its movement intersects with the line or one of its endpoints, as illustrated in the 3rd image in Figure 6. This is again done using the closest point on line function multiple times with both endpoints and the movement vector of the circle in order to test for all cases. The end result is that collision detection is done only using the closest point to a line formula, rather than more expensive, but direct calculations that calculate the bounds of the area swept by the circle. At the least that method involves vector normalization to shift the movement vector horizontally, which involves either expensive trigonometric functions or a slightly expensive square root function.

Now that a collision has been detected, it is important to figure out what the circle has collided with: an endpoint or a line? Voronoi regions are areas of points that are closer to a point or a line than any other point or line. This is determined using the information found earlier about which part of the circle collided with which part of the line. For example, if the circle's path intersects with the line and the closest points on the movement vector of the circle to each of the endpoints are both outside the movement vector, as seen in Figure 5, then the circle is in the line region. Voronoi regions for a line are shown in Figure 7. Just because the circle is in a certain voronoi region does not guarantee that it has collided with that part of the line. That requires the previous information gathered before when the collision was first detected. The exact test that determined there was a collision

can be used in conjunction with the voronoi region to determine what part of the line the circle has collided with.

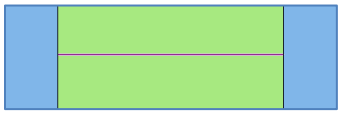


Figure 7. The green area depicts the line region for the line, while the blue areas depict the endpoint regions. It is important for endpoint regions to be differentiated a circle may intersect both endpoints, but hits one first.

The exact location of the collision can then be resolved once we know what the moving circle has collided with. If the circle has collided with an endpoint, then the closest point on the movement vector to the endpoint can be found, and then backtrack from that point until the circle only just touches the endpoint in question. A similar technique could be used for a collision between the surface of the line the circle. The method actually used involves shifting the line perpendicularly by the radius of the circle and then solving for the intersection between that line and the movement vector, giving us the location of the circle when the circle first touches the actual line.

Edge cases, seen in Figures 8 through 11, present a major problem here, especially when a circle in one voronoi region collides with a different part of the line. Other edge cases result from unusual cases when comparing the different closest points on a line with each other to determine which part of the line the circle collided with.

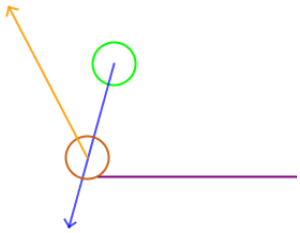


Figure 8

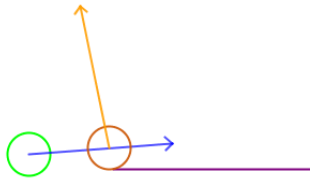


Figure 9

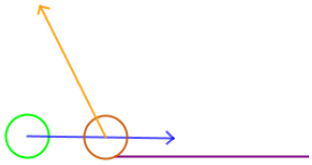


Figure 10

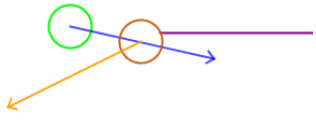


Figure 11

Figure 8 shows the side of the circle hitting the endpoint when approaching from the line side of the endpoint. This is unusual since the back of the circle collides with the line, which may cause the circle to move more parallel to the line. Upon closer analysis, the

circle is close to colliding with the line instead of the endpoint, so care is necessary to smoothly transition from one to the other as the angle of attack changes.

Figure 9 shows a similar case to Figure 8 in that the circle nearly misses the line though this time it is nearly parallel. This is tricky as it is still the front of the circle that collides with the line but the circle is projected off at an odd angle as result.

Figure 10 has a circle that slightly pointed toward the line but is nearly parallel. Here it is necessary carefully differentiate between whether the circle collides with the line or the endpoint. If it collides with the line, it would be projected strangely since it is below the line by the time it is past the endpoint. It is also necessary to prevent the circle from bouncing off the other endpoint; a problem more easily solved using distance.

Figure 11 has a circle that will not collide with the line but does collide with the endpoint, though more tightly than the case in Figure 9. This is a more common case than the previous edge cases and the main issue here is also simpler, that the endpoint collision should take priority over a line collision because it occurs earlier and is possible, even though the point of intersection with the line without the endpoints is extremely early.

Collision Response

If the circle has collided with an endpoint, then the endpoint can be treated as a circle that has a radius of zero. More simply, the magnitude of the original velocity is kept, while the direction is changed to the direction from the endpoint to the location of the collision, as seen in earlier diagrams. If the circle collided with the line part of the line, then the movement vector could just be reflected over the line normal to give a new resultant. After the new velocity is calculated, the length of the original velocity that hadn't been travelled yet is traversed and the circle is translated it its new position.

Experiment

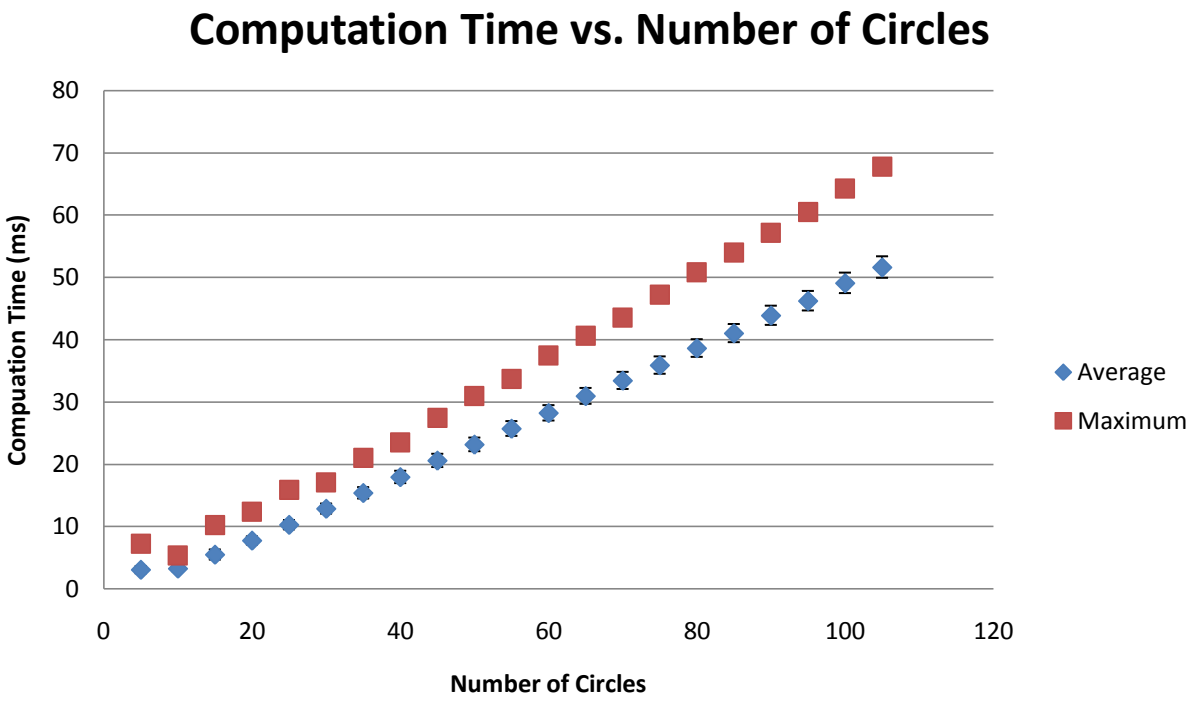
The speed of the algorithm was tested by arranging 100 circles in 50 crosses, each composed of two lines, all in a grid format. Each of the circles was given a random velocity, but they all had the same mass. The list of lines and circles was then passed to the algorithm and the computation time for each pass through the algorithm was recorded. `System.nanotime()` was used to measure the current time and the time elapsed was derived from separate measurements of the current time. Collisions off walls that were not part of the list of lines were calculated and included in the calculation time, though such collision only require a constant amount of time.

The number of circles was varied from 1 through 100 at intervals of 5 circles. This was done to show the relationship between the number of circles and computation time. Separately, the radius of the circles was varied, which changed the frequency of collisions. This is because a smaller radius, relative to the area of the field and the length of the lines resulted in a relatively larger field, which decreases the frequency of collisions. This allows for the visualization of the relationship between the frequency of collisions and computation time.

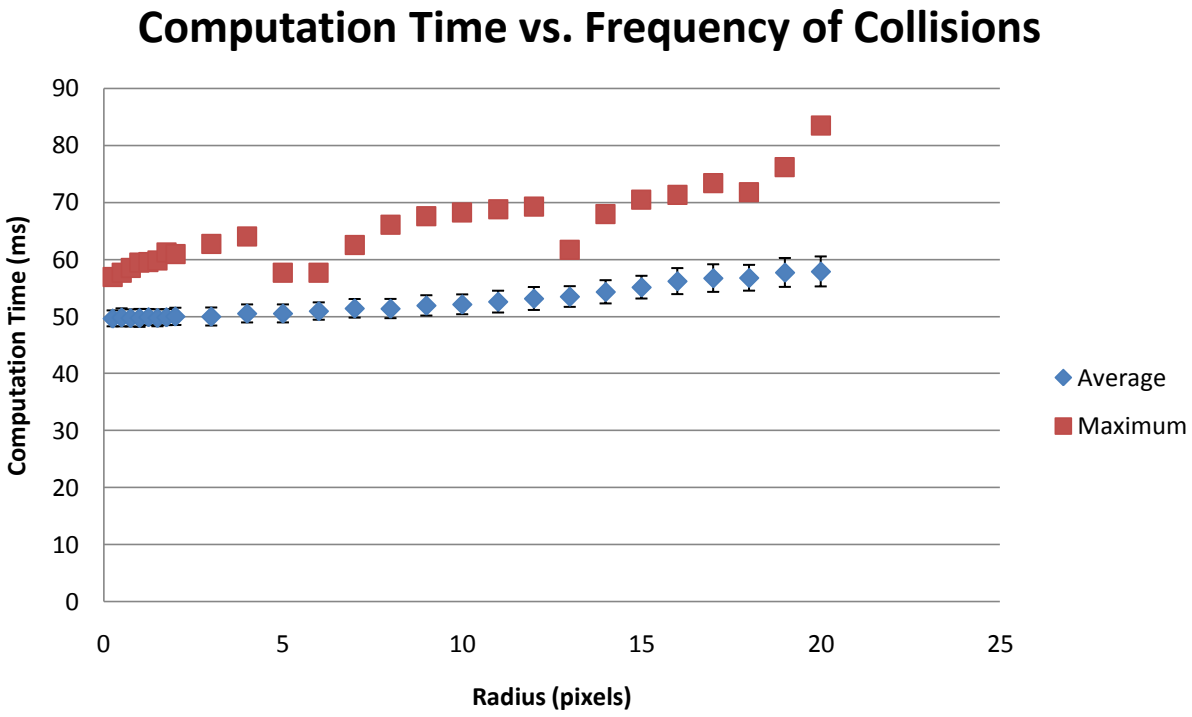
Measurements were done on an Intel Core 2 Duo E6600 with 2GB DDR2 RAM, an Nvidia GeForce 9800GT, Windows XP Professional SP3, and Java 6 Update 17. Results may vary on single core and multi-core processors, though this has not been tested.

Results

Graph 1



Graph 2



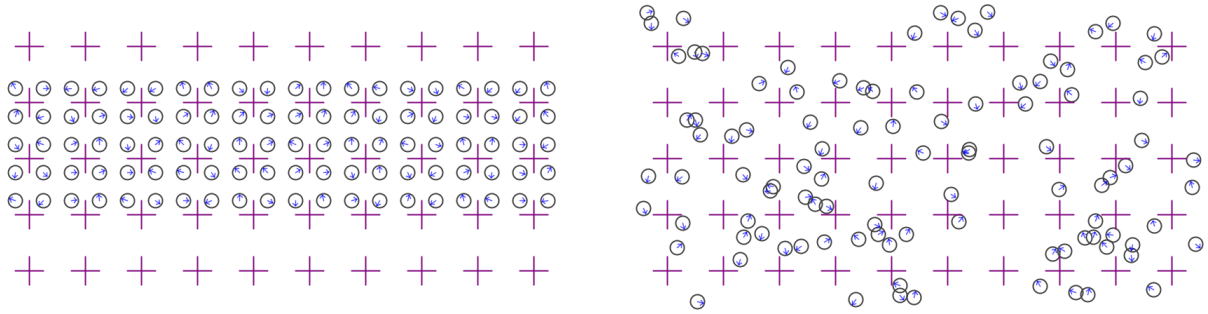


Figure 12. The left image shows the initial position of the circles and lines, and the right image shows the final position after 600 frames have passed.

Conclusion

The algorithm developed is realistic and fairly accurate, as there are no perceptible errors when it is run in real-time. Many edge cases are handled correctly, such as the unusual case where a circle hits an endpoint from behind. Yet there are still errors with the algorithm as sometimes impossible resultant angles may be calculated due to the strange positioning of the circle relative to the line. From observation, most cases are simple head-on collisions with the endpoints or with the surface of the line, not one of the trickier edge cases. The algorithm is also not time reversible due to assumptions made by the framework, though this is not a concern.

An average of 52 milliseconds, or 19.2 frames per second, was achieved with 100 circles and 100 lines, well under the target of 60 frames per second. The use of square root, a relatively expensive operation relative to algebraic manipulations has been minimized throughout the algorithm, thereby improving speed. Computation time is directly related to the number of circles, as seen in Graph 1, thus proving the $O(nm)$ claim. There is also a greater variation in the computation time as the number of circles increases, which makes sense given that the maximum computation time also increases, so there are just frames

where there are many collisions, a situation that could not have occurred with fewer circles.

Computation time is also at least slightly related to the frequency of collisions, as seen in Graph 2. Yet it also makes sense here that the standard deviation increases because the maximum computation time also increases more so than the average time, so the deviation from the mean would be higher. Also, the mean is further from 0 milliseconds, so frames with nearly no collisions, which may occur, will also increase the standard deviation. Maximum computation time is highly variable against the frequency of collisions, probably because radius is not perfectly related to the frequency of collisions.

Future Work

The algorithm, though functional, is too specific to be used alone and could be improved. First, the code could be optimized to improve running time by adding more tests so that non-colliding objects can leave the algorithm earlier. Also, duplicative tests need to be found and eliminated since much of the information obtained from an initial situation is overlapping. Also, there is an unusual edge case that involves a circle colliding with an endpoint when in fact it collides with the line first, producing a resultant angle that would be impossible to reproduce in reality. This may be a more common case than believed since a circle could collide with any part of the line. Lastly, the work presented here can serve as a basis for polygon-polygon or circle-polygon collision detection and response, a highly generalized approach to collision simulation that involves reducing all objects into simple polygon meshes. This is the prevailing method used today for 3-dimensional collision modeling.

Bibliography

Baraff, D. (1993). Non-penetrating Rigid Body Simulation. *Eurographics '93 State of the Art Reports.* Barcelona: Eurographics Association.

Canale, R. P., & Chapra, S. C. (2001). *Numerical Methods for Engineers: With Software and Programming Applications* (Fourth ed.). McGraw Hill.

Cohen, J. D., Ming, L. C., Manocha, D., & Ponamgi, M. (1995). I-Collide: An Interactive and Exact Collision Detection System for Large-Scale Environments. *Synopsium on Interactive 3D Graphics* , 189-196.

Hahn, J. K. (1988). Realistic Animation of Rigid Bodies. *Computer Graphics , 22* (4), 299-308.

Heuvel, J. v., & Jackson, M. (2002, Janurary 18). Pool Hall Lessons: Fast, Accurate Collision Detection Between Circles or Spheres. *Gamasutra* .

Hubbard, P. M. (1996). Approximating Polyhedra with Spheres for Time-Critical Collision Detection. *ACM Transactions on Graphics , 15* (3), 179-210.

Hubbard, P. M. (1995). Collision Detection for Interactive Graphics Applications. *IEEE Transactions on Visualization and Computer Graphics , 1* (3), 218-230.

Mirtich, B. (2000). Timewarp rigid body simulation. *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques* (pp. 193-200). New York, NY USA: ACM Press/Addison-Wesley Publishing Co.

Mirtich, B. V. (1996). Impulse-Based Dynamic Simulation of Rigid Body Systems. Department of Computer Science, University of California, Berkeley.

Moore, M., & Williams, J. (1988). Collision Detection and Response for Computer Animation. *Computer Graphics* , 289-298.

O'Sullivan, C., & Dingliana, J. (2001). Collisions and Perception. *ACM Transactions on Graphics , 20* (3), 151-168.

Redon, S., Kheddar, A., & Coquillart, S. (2002). Fast Continuous Collision Detection between Rigid Bodies. *Computer Graphics Forum , 21*, 279-288.

Schwager, T., & Pöschel, T. (1998). Coefficient of normal restitution of viscous particles and cooling rate of granular gases. *Physical Review E , 57* (1), 650-654.