An Approach to Accurate Multiple 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).

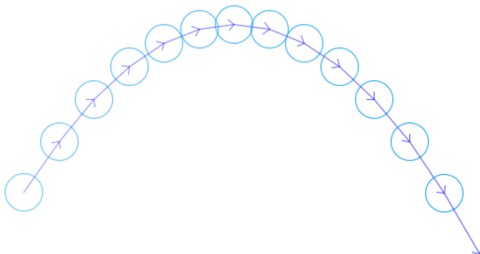


Figure 1. The movement of the circle is split up into discrete positions and velocities at certain times (increasing opacity indicates the passage of time), approximating a continuous parametric function, a parabola in this case.

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 each object, o, is stored in a list, of size n, then the list can be traversed iteratively: $o_1, o_2, o_3 \ldots o_n$. Therefore, each object is being compared to $n-1$ objects, and half of those comparisons are duplicates because of order: o_1, o_2 vs. o_2, o_1. So the running time of the function in Big O notation is

$$O\left(\frac{n(n-1)}{2}\right) \approx O(n^2)$$

This means that there is a quadratic relationship between the number of objects and the amount of time it takes to complete the algorithm (Hahn, 1988; Hubbard, 1995). It is important to minimize the running time of the collision detection function, because the value of n^2 is extremely large for large values of n and collision detection is the slowest part of the physics simulation process (Mirtich B. V., 1996; Cohen, Ming, Manocha, & Ponamgi, 1995).

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).

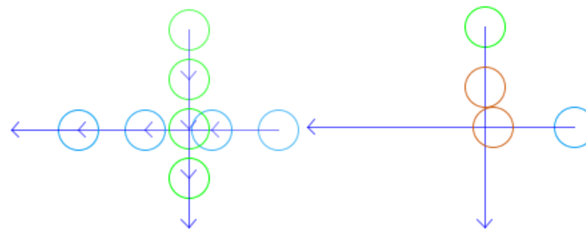


Figure 2. Diagrams generated by the collision detection algorithm discussed later in this paper, showing the results of two different collision methods. On the left, the objects are moved forward in small increments and intersections are only done between the final positions, resulting in tunneling.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

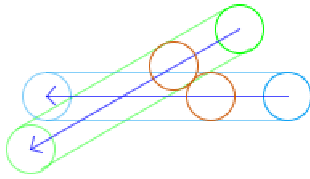


Figure 3. The green and blue circles sweep an area as they move along their movement vector. The overlapping of indicates that there is a possibility of a collision occurring between the two circles.

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 $1 - C_R$.

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

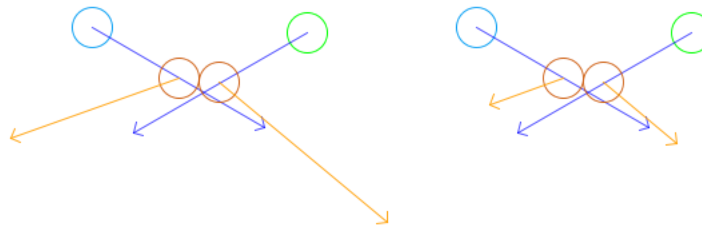


Figure 4. On the left is a perfectly elastic collision; on the right is an inelastic collision with a coefficient of restitution of .5. As a result, the resultant vector lines represented by the orange arrows are shorter in the collision on the right.

The collision response algorithm runs only when a collision actually occurs and this particular implementation is fairly simple, so speed is not of concern(Baraff, 1993). Accuracy is therefore the main concern when writing a collision response algorithm.

<u>Issues with the Single Collision Algorithm</u>

The aforementioned algorithm for collision detection and response has several weaknesses. If a collision is detected to occur before the object reaches the end of its trajectory, then the object only moves to the location of the collision because the change in trajectory requires collision detection to be done again, so an object would not travel the full length of its original movement vector. If it doesn't, the object would be moving at a different speed than it should be, as speed is distance over time. So after a collision, an

object must move the remaining distance in the new direction defined by the new, post-collision vector.

This modified algorithm still falls under the quadratic running time of the original algorithm, but simply running the original algorithm again would result in approximately double the running time, making it possibly too slow for certain purposes (Hahn, 1988). Also, any algorithm that requires true accuracy after a single collision would re-run the collision detection code after the first collision that occurs in simulation time. All subsequent collisions that were detected may not occur at all once since the movement vectors of collided objects will be modified, rendering the collisions detected based on the initial movement vectors wrong. Running the collision detection algorithm that many times is impractical for time-constrained simulations and such accuracy may be unnecessary.

The goal is to design a collision simulation algorithm that can handle multiple collisions within one frame by running again to calculate each successive set of collisions, without a large impact on running time. Previous collision algorithms have not addressed this issue and instead move on to the next frame, where collision simulation is done again.

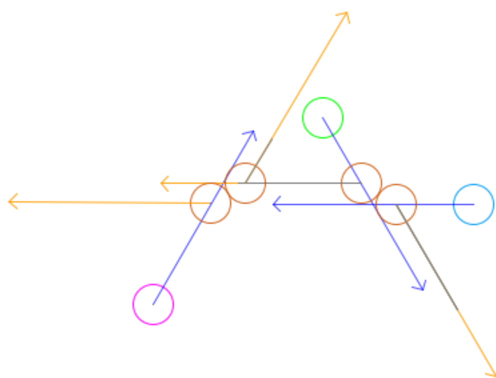


Figure 5. The colored circles represent the starting positions at the beginning of the frame, and blue lines are their initial velocities. Dark orange circles denote the location of the collision, while orange lines show the resultant velocity after the collision. The green circle collides with the blue circle then, later on but within the frame, collides with the magenta ball, which had not previously collided with anything.

Setup

The algorithm was written in Java using circles as the objects, as they are the simplest objects to do collision detection with (Hubbard, 1996). The algorithm works on a list of objects that contain a circle's position, velocity, mass, and radius. These can all be condensed into a single class, or object definition, called `Circle` that contains each of these properties. Here is a shortened version of the Java class:

```
public class Circle {
     float x, y, vx, vy, mass, radius;
     public Circle(float x, float y, float vx, float vy, float
          mass, float radius); //constructor
}
```

These circles were then stored in a list implemented `ArrayList<E>`, which uses an array to store the data. This implementation was chosen because of its $O(1)$ access time, which is important when there are at least n^2 accesses of the array by the collision detection algorithm.

The general structure of the collision simulation method, which only has a list of objects as its input, is to check for collisions before they occur using the objects movement vector, calculate the responses, and return a new list that contains each object's new location and velocity, without modifying the original list.

The algorithm needs to store several temporary pieces of information while it is running that should not be attached to the objects themselves, as they are specific to a single frame's worth of calculations. Assuming the size of the list of objects does not change, if an `array` of each type is made, then the index of the array can correspond to the index of the circle in the list of circles. One variable is the distance to the collision, or the time elapsed between the time of collision and the initial time. One can be converted to the

other using the velocity of the object. Distance squared is actually used in this algorithm because a square root is expensive, so the variable is called `closestdistsq`. Obviously, a `Point` object to store the location of the collision so that collision response can perform the appropriate calculations is also needed. This was combined with the resultant vector in to form a `PointVector` that had a new position and a velocity for each object that collided, and thus the array was named `results`.

Another variable is a list for each object to store references to the objects it has collided into, `colliders`. A reference to the object collided into, the index of the object in the list, and force of the collision are needed, wrapped in an object called `Collider`, seen here:

```
private class Collider {
    private Object collideobj;
    private Integer collideindex;
    private Vector collideforce;
}
```

A generic resizable list of colliders was implemented an ArrayList, just in case an object is involved in a collision with more than two objects. This list was then wrapped with another class:

```
private class Collideclassindex {
    private ArrayList<Collider> colliders;
}
```

Finally, this is the basic structure of the algorithm, with the variable instantiation:

```
public collide(ArrayList<ForceCircle> circles){
    PointVector[] results = new PointVector[size];
    double[] closestdistsq = new double[size];
    Collideclassindex[] collisionindex = new
        Collideclassindex[size];
    double[] collisiontimes = new double[size];
    collision_detection();
```

```
            collision_response();
            return new Collideresult(results, collisionindex,
                  collidedwith);
      }
```

If collisions are assumed to not be occurring near each other, then each collision is independent, with no effect on any another collision. The results of collisions occurring at different times can be calculated during the initial run-through of the list of objects. The pair-processing algorithm used is *a priori*, collision detection before it occurs.

There are three basic cases that may occur when checking for collision between two objects: they may both be moving, one of them may be moving, or they are overlapping. If they are both moving, the collision detection can be simplified by using the reference frame of one of the objects so that one object remains still and the other object is moving (Heuvel & Jackson, 2002). This reduces a more complex problem into the simpler moving-vs.-non-moving problem, which does not involve time. Lastly, the problem of both objects not moving is simplest: check for intersection between the two objects, which may occur due to floating point rounding error from the finite numbers stored in computers resulting in two objects intersecting when they are just touching each other. This may result in a miscalculation with the other methods, such as two objects moving away from each other being found to collide, so it is best to resolve the intersection.

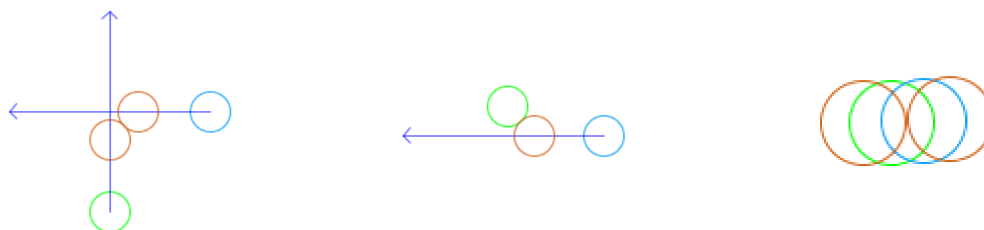


Figure 6. These diagrams show the collision possibilities between two circles, as generated by the collision detection algorithm. The heavy green, blue, and magenta circles are the initial positions of different circles, and the orange circles are the detected locations of the collision. The blue line is the initial movement vector. The first image is a moving-moving circle collision problem, the second is a moving–non-moving circle collision problem and the third is a non-moving–non-moving circle collision problem.

If both objects are moving, one object is chosen to be the reference frame so the problem is simplified to collision detection between a moving and a non-moving object (Hahn, 1988). When the area of the non-moving object overlaps with the area of the moving object, a collision has definitely occurred, the location of which can be found using a method by Heuvel and Jackson (Heuvel & Jackson, 2002). Then, the distance from the initial starting point to the location of the collision and the location of the collision are stored. The objects involved in the collision each store a reference to each other so that the collision response code can calculate the resultant velocity for each object.

Even if a collision is detected to occur within a frame between two objects at a certain point, the path of either object may have been modified by an earlier collision that has not been detected yet because of object's position in the list. This is resolved by checking for all collisions and only keeping the ones that occur first for each object. A collision occurs earlier in the simulation than another collision when the distance to the object from its starting point to the point of collision is shorter than the distance to the other collision. This frequently occurs when a collision that had been found earlier is found to not actually occur later, after more collision checks are done.

The Multiple Collision Algorithm

After using the initial set of object locations and velocities to detect collisions and verifying that those collisions are the first to occur for each object, it is assumed that those collisions will occur. In reality, after the first collision that occurs in simulation time, every collision that had been detected in the first set of calculations may be affected and thus should be recalculated. This is prohibitively expensive in real time simulations, so the

assumption is that each collision is far enough away from other collisions that they can be treated independently.

Objects are expected to move a certain distance per frame in order to maintain their velocity, but in the single collision model, after a collision the final location of the object is the location of the collision.

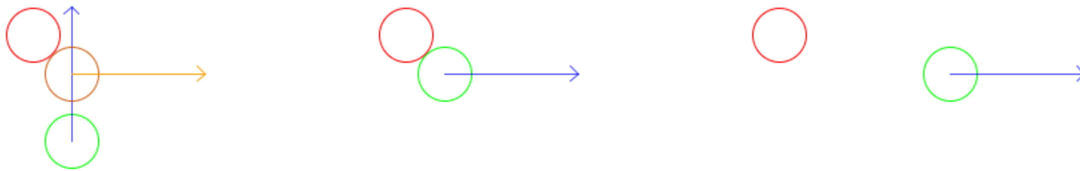


Figure 7. 3 frames of the collision simulation. The red circle is static and the green circle is moving – increasing opaqueness depicts forward movement of time. The green circle moves initially moves distance *d* over one frame, but moves 2.5*d* over the course of 3 frames because it moves only .5*d* the first frame. If *t* is the length of one frame, this means that the original velocity of $\frac{d}{t} = \boldsymbol{v}$ is actually $\frac{2.5d}{3t} = \mathbf{.83}\boldsymbol{v}$.

This means that the object moved less distance in a set amount of time than it should have, as seen in Figure 7. But the direction and speed of the object changed midway through the frame, so the previous collision checks are no longer valid. Yet at the same time, the object does not move the full length of its movement vector because it already has spent time moving along its old one. The modified collision detection algorithm must take into account the length of the original collision vector and the velocity change midway, while not recalculating collision detection between objects that have not collided yet. It must be capable of being run multiple times in order to detect successive collisions for each object, including the first collision – this is to avoid nearly doubling the amount of code in the algorithm.

Several new pieces of data are needed in order to take time into account in the collision detection. A variable, `timepassed`, is needed for each object that stores the time from the initial location of the object at the start of the frame to the object's current location and starts with a value of zero and has a maximum value of one, the full length of the movement vector. Time is stored rather than distance because the velocity of the object may change after the first collision. Also, another value, `collidetime`, is needed for each object in order to store the amount of time from the initial location – which may be the location from the last collision – to the final position of the object, which may either be the end of the vector or the location of the collision that it has been detected to be in. And finally, a `boolean`, `modified`, to store whether or not the movement vector of an object has been modified. This will have a value of `false` in the beginning of the frame, before the collision simulation algorithm is run, and is switched to `true` if a collision changes the velocity of the object. This value is used when the collision detection is done a second time.

These new variables are also temporary and yet must also be passed to the collision simulation method because the method is now being run multiple times with different conditions, or arguments:

```
public collide(ArrayList<ForceCircle> circles, double[]
      timepassed, boolean[] modified){
      //collision simulation code here
}
```

The first run of the algorithm should be no different, though the algorithm will need a helper method in order to create the new variables, as said earlier:

```
public Collideresult collide(ArrayList<ForceCircle> circles){
     double[] time = new double[circles.size()];
     boolean[] collided = new boolean[circles.size()];
     for(int i = circles.size() - 1; i >= 0; i--)
          time[i] = 1;
     return checkcollision(circles, time, collided);
}
```

The ratio of the time elapsed to the location of the collision for each object is calculated by:

$$timepassed = \frac{\sqrt{closestdistsq}}{|\vec{v}|}$$

The actual time passed would be $t_{actual} = timepassed \cdot t_{frame}$, but since the actual length of a frame is arbitrary and depends on how fast the calculations are, *timepassed* is used instead. If a collision occurs, the value of *modified* for the objects involved is flipped to `true`.

The helper method must generate a new list of objects by modifying the original list of objects with the new positions and velocities. Only modifying the objects that have `modified` set to `true` makes the process more efficient. This list is then passed to the second run through the algorithm. But first, the original collision detection algorithm must be modified in order to keep in mind the new endpoint of the collision, which is separate from the endpoint of the movement vector. This is done by a simple linear transformation for each axis, x in this example:

$$x_f = x_i \cdot (1 - timepassed)$$

Thus, the collision detection code stays within the bounds of the frame and the length of the original movement vector. Objects that have not been modified are not compared with other objects that have not been modified, since no collision had been detected between

them. The collision response code need not be changed because the new velocity is already part of the object.

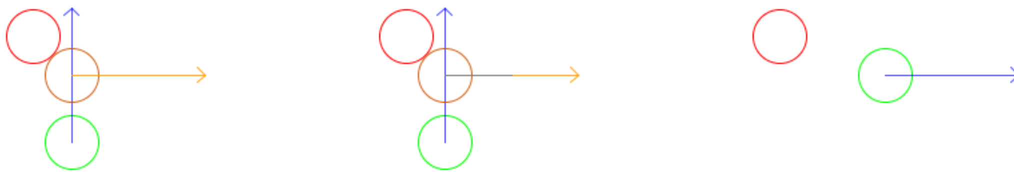
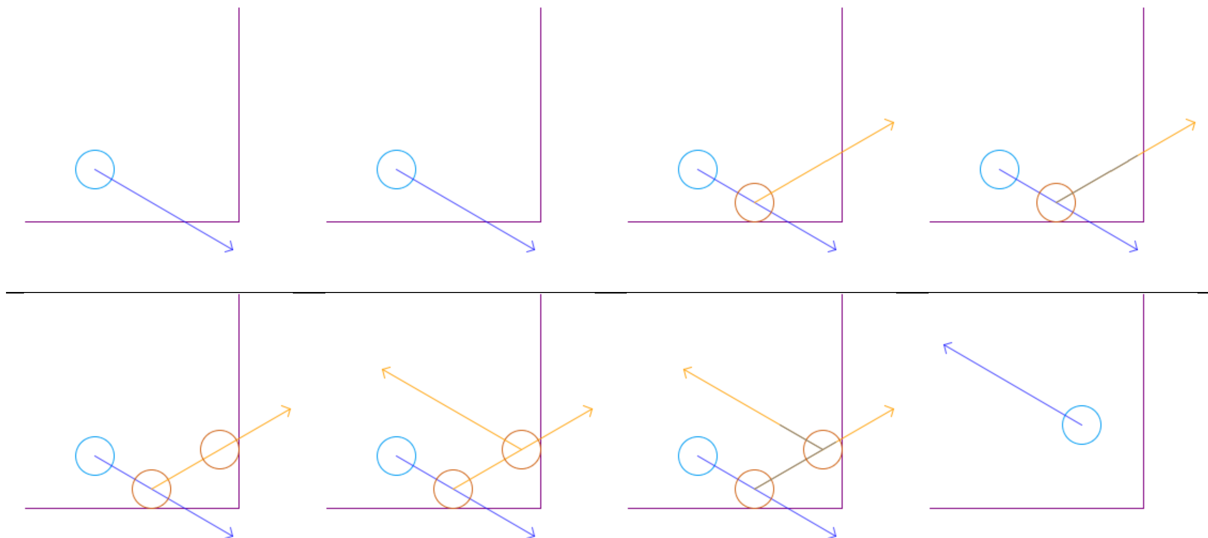


Figure 8. The multiple collision simulation algorithm. The red circle is static while the green circle is moving with a velocity shown by the blue arrow. The green circle collides with the red circle, as depicted by the orange circle, and gains a new velocity depicted by the orange arrow. Yet the circle has yet to travel the remaining 50% of the distance it was supposed to, so the collision detection code is run again using the new velocity, but only up to the distance shown by the gray line. No collision is detected, so the green object can be moved to the end of the gray line. The first two diagrams are one frame, so the circle moves distance *d*, instead of .5*d* in Figure 7. This way the circle maintains its velocity.



Timeline 1 – Multiple Collisions within a Frame. The light circle is moving, as indicated by the blue vector, into the static purple lines. The lines are stored as two independent objects, so collision detection is done independently for both of them. The orange circle is the resultant position and the orange arrow the resultant velocity. The gray line represents the length of the original movement vector left to travel.

Results and Discussion

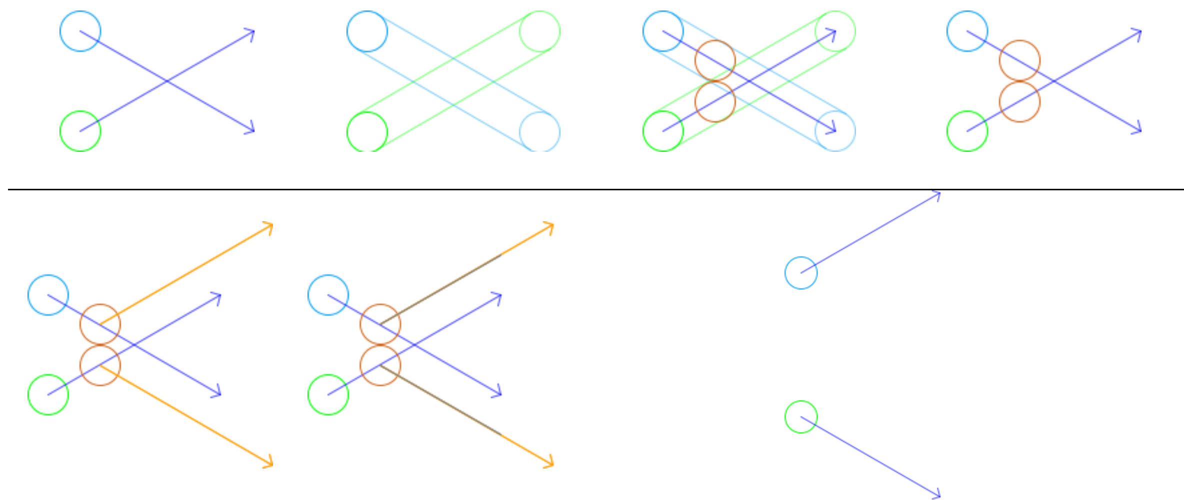
The main problem with this method is that collisions occur when they should not and collisions are missed when they should occur. This is a result of the original assumption that all collisions are independent when they are not. Yet the original single collision algorithm makes the assumption that objects stay put after a collision until the frame was over, a quite egregious error. The multiple collision algorithm was significantly more accurate than the original one, as collisions that occurred early in a frame for a fast moving object would basically appear to slow the object drastically before it instantaneously regains its speed again. This is most apparent when many objects collide into each other in rapid succession, resulting in many objects appearing to “stop” and clump together due to their lack of movement. The multiple collision algorithm allows objects to continue moving, keeping the velocity uninterrupted and the objects more spread out due to the rebound displacement.

The two algorithms were compared by putting 150 circles, arranged into 10 by 15 rows into a rectangular area with elastic walls. The simulation was then run for 600 frames on a Core 2 Duo E6600 with 2GB RAM and JDK Version 6 with Update 17 and the time to run the simulation for each frame was totaled, giving the averages seen in Table 1. The simulation was then run again, but the circles had a smaller radius, and thus a smaller area, which reduces the frequency of collisions.

Algorithm	**Avg. Time per Frame** (ms)	**Avg. Time per Frame w/ 33% original radius** (ms)
Single Collision	24.03	22.30
Multiple Collision (run twice)	36.77	31.36

Table 1

Though the speed impact of multiple collision algorithm is not negligible as seen in Table 1, it is better than running the single collision algorithm twice with its deficiencies. The time elapsed per frame is dependent on the frequency of collisions, which makes sense for both the single and multiple collision algorithms, since when there is a collision, the location must be found and the response must be calculated. Yet the percentage increase in time due to the multiple collision algorithm decreases when frequency of collisions decreases since more active or more crowded objects are more prone to secondary collisions. So if no collisions occur, a secondary collision cannot occur, while if many collisions occur near each other, it is likely that secondary collisions will occur, like when shaking a jar of jelly beans. Yet the multiple collision algorithm lowers the speed the most when it is most crucial, a significant drawback to the algorithm, though the algorithm is somewhat flexible since running it once results in the single collision algorithm. So if the time allotted is used up, then accuracy can be forsaken for speed.



Timeline 2 – Multiple Collision Simulation algorithm. The colored circles are the objects involved in the collision. The gray line represents the length of the original movement vector left to travel. The orange circles are the detected collision locations and the orange arrows are the resultant velocities.

Conclusion

This algorithm is most applicable to situations where there are a few fast moving objects among slow moving ones. A fast moving object may collide off multiple objects without a significant drop in speed because the frequency of collision between the slower moving objects is much lower.

The primary application of this research will be for real-time simulations, such as video games and training simulations. It allows for fast-moving projectiles, such as bullets, to be reliably simulated within the same framework as slower moving objects, such as people, instead of the traditional hitscan-based, or line-based, techniques, where the objects are assumed to move instantaneously in a straight line. Drag and gravity have significant effects over long distances, which will be observed with the aforementioned method. Also, the traditional technique of freezing objects that are immobile will no longer be necessary, resulting in physically realistic piles. The implementation of this algorithm will have a significant impact on the realism of physics in games, while only making a small speed impact.

Future Work

This algorithm as it stands now needs further refinement to prevent the unusual cases of collisions being created that should not exist because the physical resultant vectors match up while the times do not. This may be done by making the vectors a series of continuous movements in space, with each change in velocity marked down in time. Later on, integration with other collision simulation techniques, such as binary space partitioning, will make the algorithm much faster.

References

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.

Richard, P., Birebent, G., Coiffet, P., Burdea, G., Gomez, D., & Langrana, N. (1996). Effect of frame rate and force feedback on virtual object manipulation. *Presence: Teleoperators and Virtual Environments , 5* (1), 95-108.

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.