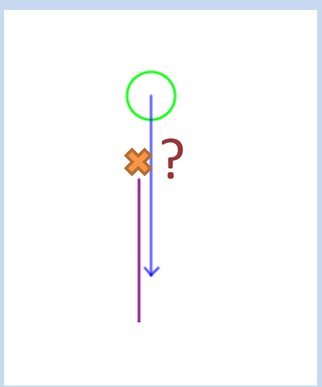
Introduction

- Physical simulations
  - Predictable and deterministic (Mirtich B., 1996)
  - Movement is split up into discrete units for speed reasons (Mirtich B., 2000)
- Objects in Simulations
  - Contain positions and velocities (Hubbard, 1995)
  - Stored in a list in random order (Hubbard, 1995)
- Static Objects
  - Are immobile
  - Energy that would have moved the object is reflected and returned to object that hit it
- Collisions
  - Collision detection and response are intertwined (Moore & Williams, 1988)
  - Objects must be compared in pairs (Hahn, 1988)

Collision Detection

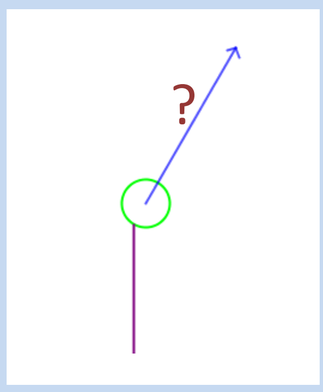
Check if a collision occurs between two objects.



1. Create a function that describes the distance between the objects over time (Hubbard, 1993)
2. Solve for when the objects are less than their radii apart from each other (Heuvel & Jackson, 2002)
3. If there is no solution, there is no collision

Collision Response

Calculate the resultant velocities.



- Obey the Law of Conservation of Momentum
- Obey the Law of Conservation of Energy
- Coefficient of restitution (C_R)
- Each axis is independent

Problem

How do you resolve collisions between dynamic circles and static lines?

- Lines have zero thickness and a finite length
- Circles sweep an area as they travel
- Endpoints must be treated differently from the line itself

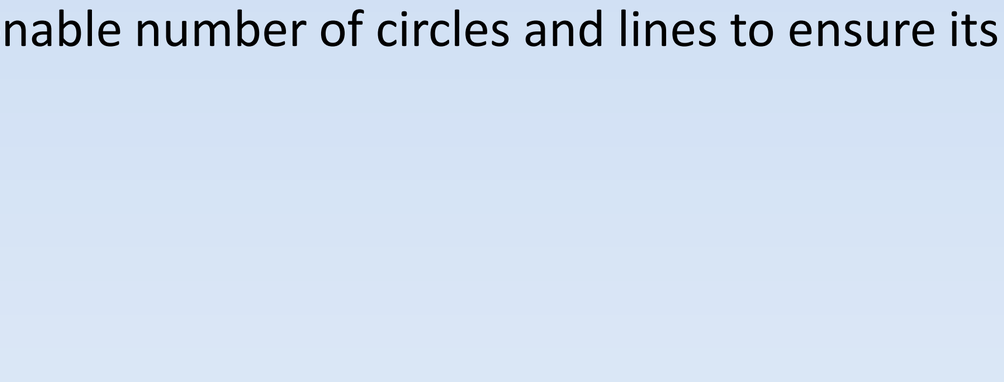
Goal

- Create a collision simulation algorithm using Java 1.6 to that can handle correctly collisions between static lines and dynamic circles
- Algorithm should run at an acceptable speed with a reasonable number of circles and lines to ensure its practicality
  - 100 moving circles with a radius of 10 pixels
  - 100 static lines with a length of 20 pixels
  - 60 frames per second

Code Appendix

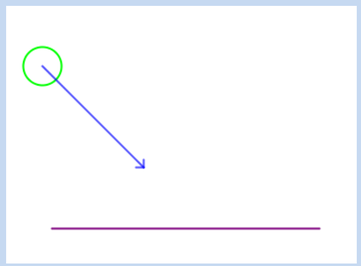
- Finding the closest point on a line to an input point is useful for determining how close a point and a line are to each other
- Below is a function that returns the closest point on a line given a line and a point, as implemented and used in my algorithm

```
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);
}
```



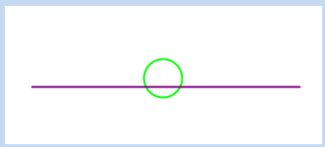
- Each dark line and dot represent one set of arguments passed to the `closestpointonline` function
- The bright line connects the input point and the closest point on the input line

Algorithm



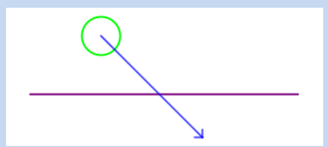
1. **Start with a moving circle and a non-moving line.**

2. **Check to see if a collision has occurred at all.**



Check if the circle has already intersected with the line.

1. Find closest point on the line to the center of the circle
2. If distance between point and center of circle is less than the radius, there is a collision

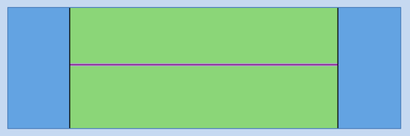


Check if the path of the circle intersects with the line.

1. Find the point of intersection between the movement vector of the circle and the line, assuming they extend to infinity
2. If point of collision is between endpoints of both lines, there is a collision

Check if the area swept out by the circle by its movement intersects with the line.

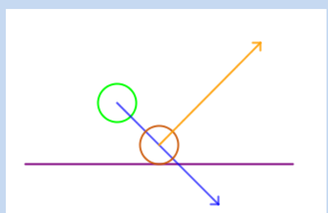
3. **Figure out the location of the circle relative to the line using voroni regions.**



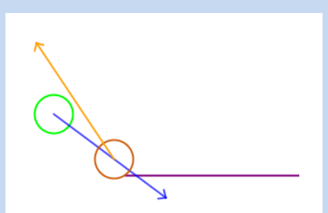
4. **Resolve the collision by determining the exact time and location of the collision.**

Whether the circle collides with the line or an endpoint determines the course of action. (*resultants shown for clarity*)

1. Linear algebra, or matrices, are used to calculate the point of intersection between two lines.
   - The movement vector, or single-frame velocity of the circle is treated as a line segment for the linear algebra.
2. If the point of collision is between the endpoints of the line, the line segments intersect
3. The closest point on a line to a point formula is used to determine if the circle collides with the endpoint of the line.
4. If the circle collides with an endpoint, determine which endpoint it collided into.

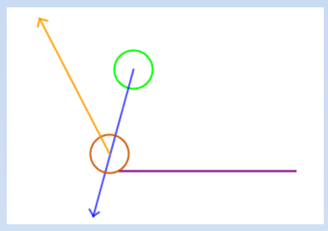


•If the path of the circle intersects with the line and does not go near an endpoint, then it is a simple surface collision

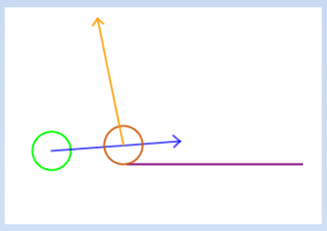


•If the path of the circle contains an endpoint and intersects with the line, then it is an endpoint collision

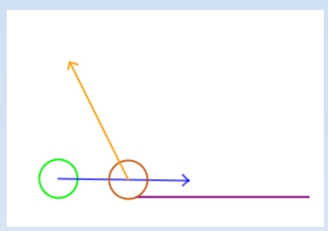
Edge cases



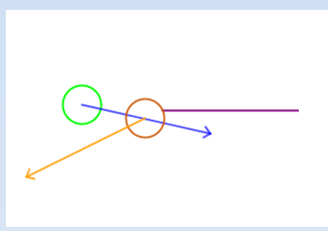
•The side of the circle may collide with the endpoint when approaching from the line side of the endpoint.



•The movement vector does not intersect with the line, but the circle collides with an endpoint.



•When the angle is shallow, the circle may intersect with the line at a point behind the circle's movement vector.



•The circle may cross the line at a point beyond the endpoint of the line.

5. **Calculate resultant velocity of the circle.**

Along each axis, these laws are used to calculate the resultant velocity:

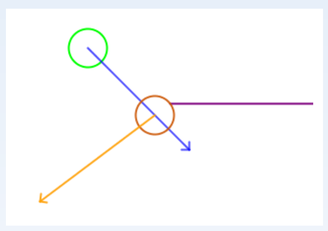
Conservation of Momentum (p) $\quad mv_i = mv_f$

Conservation of Energy (E) $\quad \frac{1}{2}mv_i^2 = \frac{1}{2}mv_f^2$

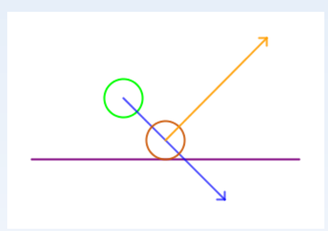
Which gives us this optimized formula for circle-endpoint collisions:

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

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

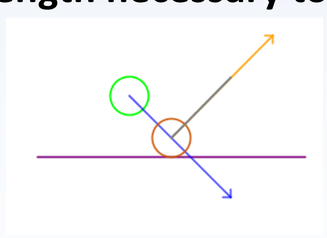


If the circle collided with an endpoint, treat the endpoint as a static circle with zero radius.

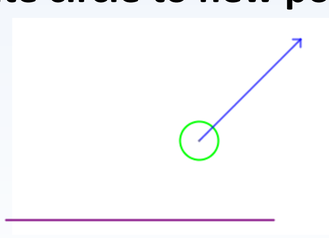


If the circle collided with the side of the line, reflect the movement vector over the normal to the line.

6. **Travel remaining length necessary to match velocity.**



7. **Translate circle to new position.**

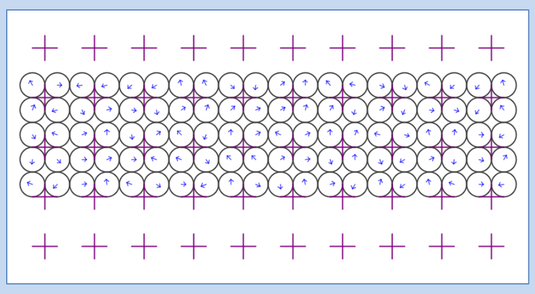


Experiment

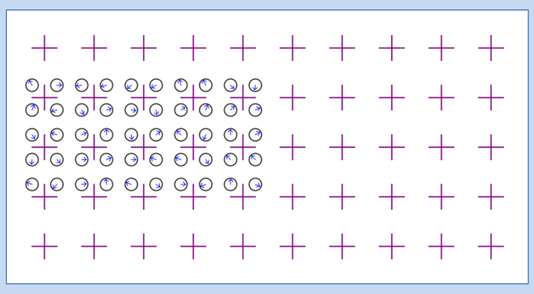
1. 100 lines arranged in 50 crosses each composed of 2 lines
2. Circles then placed with random velocities
   - Number of circles varied from 1 – 100
   - Radius of circles varied from 1 – 20 pixels, varying the frequency of collisions
3. List of lines and circles passed to algorithm
4. Computation time for each pass through the algorithm recorded
   - `System.nanoTime()` used to measure the current time
   - Collision off walls calculated and included in time elapsed, but not included in number of lines
5. Algorithm run for 600 frames

Test Computer Specifications

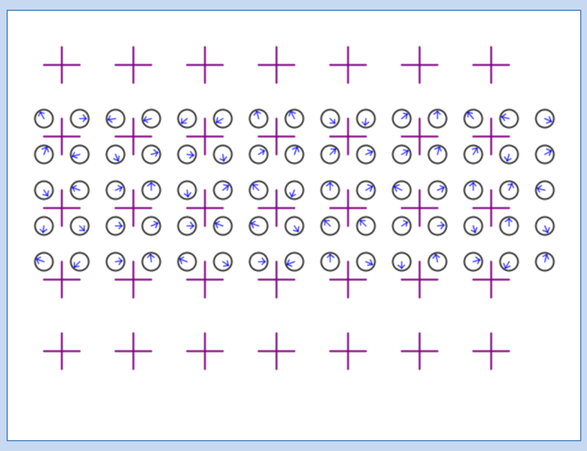
- Intel Core 2 Duo E6600
- 2 GB DDR2 RAM
- Nvidia GeForce 9800GT
- Windows XP Professional SP3
- Java 6 Update 17



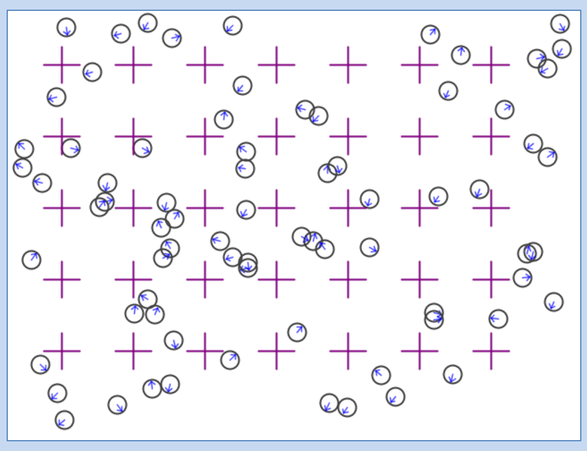
Large radius



Low number of circles

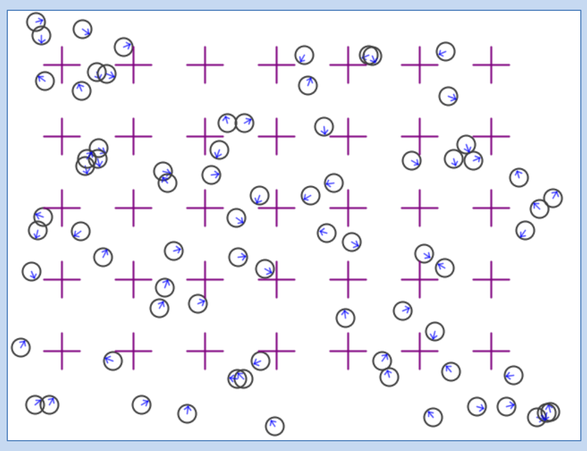


Initial State



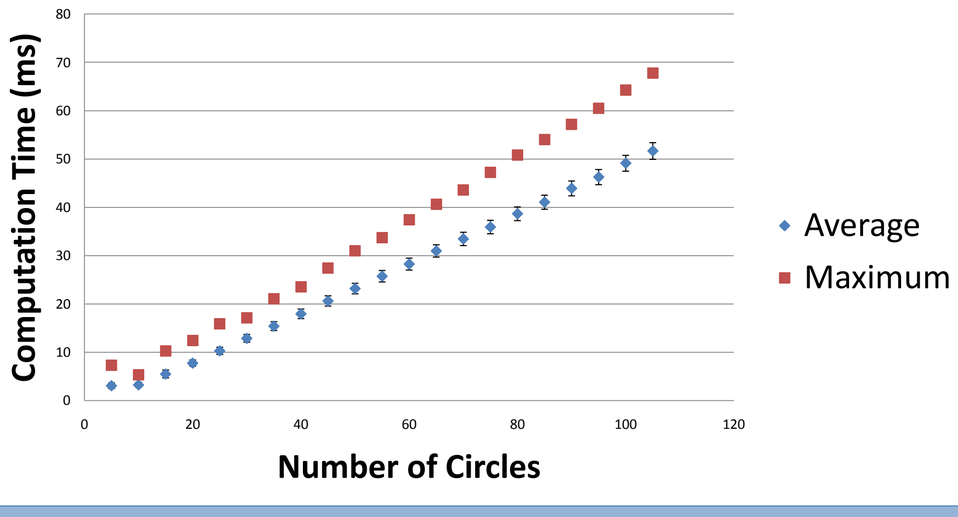
Frame 300

No collision between circles

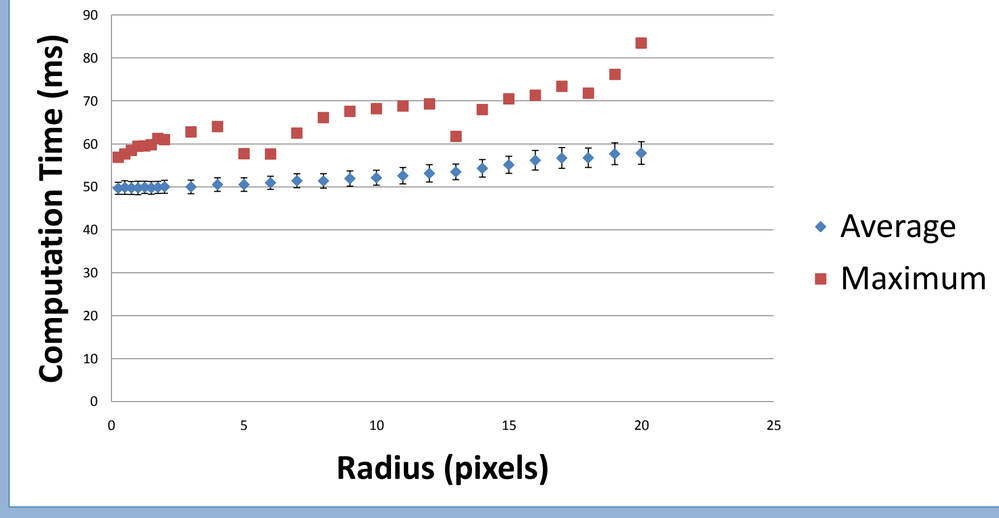


Final State

Results



- Radius of 10 pixels used



- 100 circles used

Conclusion

- Algorithm is realistic and accurate
- Edge cases handled correctly
- Use of the square root function is minimized by using linear algebra
- Computation time is directly related to number of circles, thus algorithm is $O(nm)$
- Greater variation in computation time as number of circles increases
- Maximum computation time increases as the number of circles increases
- Average of 19.2 frames per second with 100 circles and 100 lines, well under the target of 60 FPS with
- Algorithm is not time reversible
- Computation time has a slight relationship to the frequency of collisions
- Average computation time varies more as the frequency of collisions increases
- Maximum computation time is highly variable against the frequency of collisions

Future Work

- Optimize to reduce running time
- Resolve an unusual edge case (Figure 1)

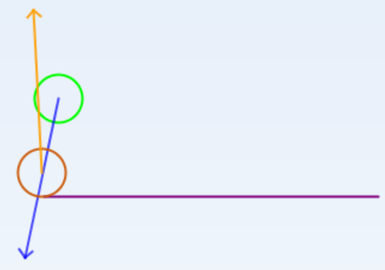


Figure 1

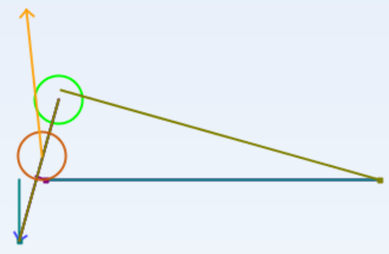


Figure 2

Works Cited

- 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. (1995). Collision Detection for Interactive Graphics Applications. IEEE Transactions on Visualization and Computer Graphics , 1 (3), 218-230.
- Hubbard, P. M. (1993). Interactive Collision Detection. Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality, (pp. 24-31). San Jose, CA, USA.
- 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.