Matt Riley
Rigid Body Particle System

Introduction

To write this rigid body particle system, I used Processing 3.0 with Java. No additional plugins were used.

Code

```
ParticleSystem ps;
int numCollisions = 0;
int numParticles = 100;
int particleLifetime = 255;
void setup() {
  size(640,360);
  ps = new ParticleSystem(new PVector(width/2,50));
}

void draw() {
  background(0);
  ps.run();
  if(ps.particles.size() < numParticles) {
       ps.addParticle();
  }
}
```

To start, we declare a global ParticleSystem that will house all of our particles. Additional global variables to control properties of the system are declared here as well. This includes the current number of collisions, the limit of the number of particles, and the lifetime of the particles.

Processing calls the setup() function exactly one time, where we create a 640 by 360 pixel scene, and create a new ParticleSystem instance (with an arbitrary position of the middle of the screen - individual particles are not limited to this position).

The draw() function is then executed on every frame. We draw our black background and call the run() function that's in our ParticleSystem class. In addition, we draw a new particle if the limit of particles has not been reached yet.

```
// A class to describe a group of Particles
// An ArrayList is used to manage the list of Particles

class ParticleSystem {
  ArrayList<Particle> particles;
  PVector origin;

  ParticleSystem(PVector location) {
    origin = location.get();
    particles = new ArrayList<Particle>();
  }

  void addParticle() {
    particles.add(new Particle (new PVector(floor(random(0, 600)), floor(random(0, 400)))));
  }

  boolean detectCollision(PVector b, PVector c, Particle id) {
  float dx = c.x - b.x;
  float dy = c.y - b.y;
  float distSquared = dx * dx + dy * dy;

  if (distSquared < (id.r * id.r)) {
    numCollisions += 1;
    return true;
  }
  else {
    return false;
  }
}
```

Here we have our class definition of a ParticleSystem. There are two fields, an ArrayList of particles, and the origin of the system (which, as mentioned before is fairly arbitrary).

In addParticle() we push a new Particle instance onto the ArrayList.

detectCollision() is used to check if there has been a collision between the two position vectors ‘b’ and ‘c’ (while also taking into account the radius of the Particle we’re currently dealing with).

```
  void run() {
    for (int i = particles.size()-1; i >= 0; i--) {
      Particle p = particles.get(i);
      p.run();
      for (int j = i - 1 ; j >= 0; j--) {
        Particle c = (Particle) particles.get(j);
        if(detectCollision(p.getPosition(), c.getPosition(), c)) {
          p.didCollide(c);
        }
      }

      if ((p.isDead()) || (p.location.y > 360)) {
        particles.remove(i);
        this.addParticle();
      }
    }
  }
}
```

This run() method inside ParticleSystem is called on every frame by our draw() method. We iterate through the ArrayList of particles, and then call the run() function inside the Particle class, as well as check for collisions between the current particle and then all the particles that come after this (we can do this because of the nature of collisions, as

a collision between A and B is also indicative of a collision between B and A - so we don't want to check if A collided with B AND if B collided with A - better to just check one). If there has been a collision, we delegate that to the didCollide() method inside Particle to set the new velocity vector for both particles.

If a particle's lifetime is up or if it's fallen off the screen, we remove it from the ArrayList and create a new one.

```
// A simple Particle class

class Particle {
  float mass;
  PVector location;
  PVector velocity;
  PVector acceleration;
  float lifespan;
  boolean collided;
  float r;
  float m;

  Particle(PVector l) {
    mass = random(1, 100);
    acceleration = new PVector(0,0.05);
    velocity = new PVector(random(-1,1),random(-2,0));
    location = l.get();
    lifespan = particleLifetime;
    r = random(4, 16);
    m = r*.1;
  }

  PVector getPosition() {
   return this.location;
  }
```

The inside of our Particle class. We have a mass (which is random at the moment but can be changed), acceleration, velocity, position (named location), a lifespan, and a radius r.

```
void didCollide(Particle that) {
  // get distances between the particle's components
  PVector bVect = PVector.sub(that.location, location);

  // calculate magnitude of the vector separating the particles
  float bVectMag = bVect.mag();

  if (bVectMag < r + that.r) {
    // get angle of bVect
    float theta  = bVect.heading();
    // precalculate trig values
    float sine = sin(theta);
    float cosine = cos(theta);

    /* bTemp will hold rotated particle's positions. You
     just need to worry about bTemp[1] position*/
    PVector[] bTemp = {
      new PVector(), new PVector()
      };

      /* this particle's position is relative to the other
       so you can use the vector between them (bVect) as the
       reference point in the rotation expressions.
       bTemp[0].position.x and bTemp[0].position.y will initialize
       automatically to 0.0, which is what you want
       since b[1] will rotate around b[0] */
      bTemp[1].x  = cosine * bVect.x + sine * bVect.y;
      bTemp[1].y  = cosine * bVect.y - sine * bVect.x;
```

If particles have collided, we have to calculate new velocity vectors for both of them. We find the distance between the position vectors, convert it to a magnitude, and then calculate a temporary array of vectors to hold the new position of the particles after the collision.

```
// rotate Temporary velocities
PVector[] vTemp = {
  new PVector(), new PVector()
  };

vTemp[0].x  = cosine * velocity.x + sine * velocity.y;
vTemp[0].y  = cosine * velocity.y - sine * velocity.x;
vTemp[1].x  = cosine * that.velocity.x + sine * that.velocity.y;
vTemp[1].y  = cosine * that.velocity.y - sine * that.velocity.x;

/* Now that velocities are rotated, you can use 1D
 conservation of momentum equations to calculate
 the final velocity along the x-axis. */
PVector[] vFinal = {
  new PVector(), new PVector()
  };

// final rotated velocity for b[0]
vFinal[0].x = ((m - that.m) * vTemp[0].x + 2 * that.m * vTemp[1].x) / (m + that.m);
vFinal[0].y = vTemp[0].y;

// final rotated velocity for b[0]
vFinal[1].x = ((that.m - m) * vTemp[1].x + 2 * m * vTemp[0].x) / (m + that.m);
vFinal[1].y = vTemp[1].y;
```

We then determine the new velocities of each particle. We have to rotate the current velocities of the particles so that we're dealing with the angles that the particles actually meet at - so we can used 1D kinematics equations. Afterwards, we can calculate the final velocity vectors of both particle A and B using conservation of momentum equations.

```
  /* Rotate particle positions and velocities back
   Reverse signs in trig expressions to rotate
   in the opposite direction */
  // rotate particles
  PVector[] bFinal = {
    new PVector(), new PVector()
    };

  bFinal[0].x = cosine * bTemp[0].x - sine * bTemp[0].y;
  bFinal[0].y = cosine * bTemp[0].y + sine * bTemp[0].x;
  bFinal[1].x = cosine * bTemp[1].x - sine * bTemp[1].y;
  bFinal[1].y = cosine * bTemp[1].y + sine * bTemp[1].x;

  // update particles to screen position
  that.location.x = location.x + bFinal[1].x;
  that.location.y = location.y + bFinal[1].y;

  location.add(bFinal[0]);

  // update velocities
  velocity.x = cosine * vFinal[0].x - sine * vFinal[0].y;
  velocity.y = cosine * vFinal[0].y + sine * vFinal[0].x;
  that.velocity.x = cosine * vFinal[1].x - sine * vFinal[1].y;
  that.velocity.y = cosine * vFinal[1].y + sine * vFinal[1].x;
}
```

Now that we've finished our final velocity calculations, we have to convert the velocities back into positions that make sense in this 2D space. We just rotate them back by backtracking on the equations we used to rotate them in the first place. Afterwards, we just update the velocities of both particles to the calculated final velocities.

```
void run() {
  update();
  display();
}

// Method to update location
void update() {
  velocity.add(acceleration);
  location.add(velocity);
  lifespan -= 1.0;
}

// Method to display
void display() {
  stroke(255,lifespan);
  fill(255,lifespan);
  ellipse(location.x,location.y,(r*2),(r*2));
  text("Number of collisions: " + numCollisions, 100, 100);
}

// Is the particle still useful?
boolean isDead() {
  if (lifespan < 0.0) {
    return true;
  } else {
    return false;
  }
}
```

Finally, we have our run() method that's called on every frame. Update() just moves the location based on Newtonian physics of position, velocity, and acceleration, as well as decrementing the lifespan of the particle.

We also have a method to actually draw the particle, which consists of a call to stroke() (which sets the outline color), a call to fill() (the fill color of the shape, something we can easily manipulate, and the transparency is determined by the lifetime left in the Particle), and then a call to draw the shape. We currently have it drawing a circle, but we can swap this out for a method that draws a polygon by implementing a global function that makes one, like:

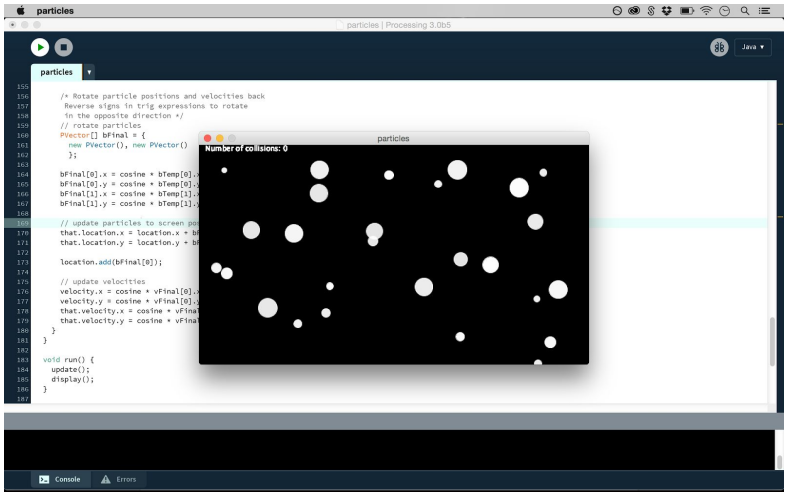
```
void polygon(float x, float y, float radius, int npoints) {
  float angle = TWO_PI / npoints;
  beginShape();
  for (float a = 0; a < TWO_PI; a += angle) {
    float sx = x + cos(a) * radius;
    float sy = y + sin(a) * radius;
    vertex(sx, sy);
  }
  endShape(CLOSE);
}
```

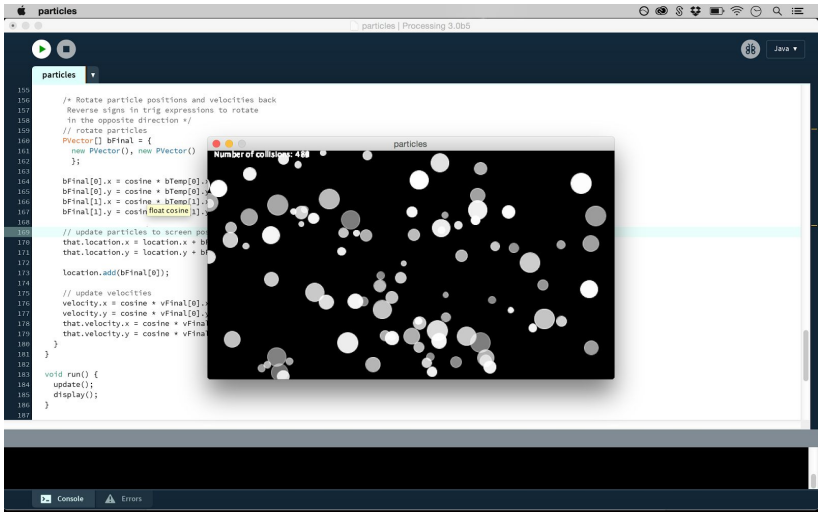
Then we can replace the ellipse() call with polygon(location.x, location.y, r, 3) to draw a triangle, for example.

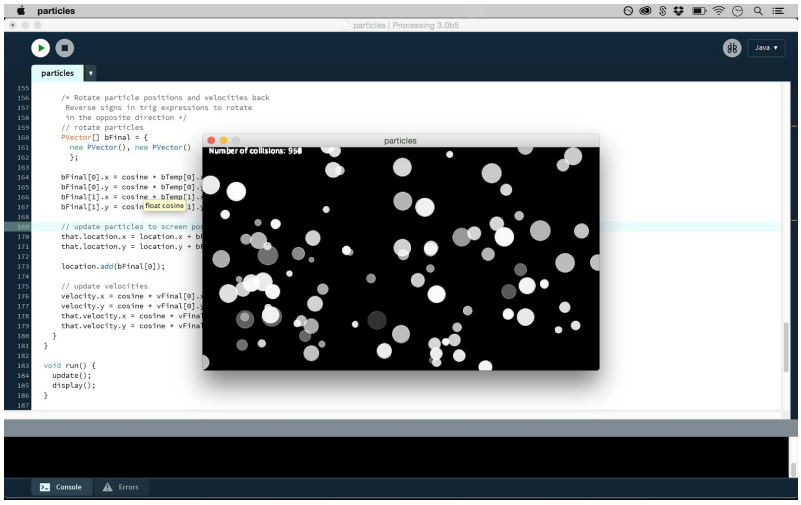
There's also a method in Particle that flips the boolean of whether the Particle's lifespan has run out or not.

Results

The results were pretty successful. Below are some screenshots:







Obviously it's a little hard to determine whether the collisions are working just from screenshots; the Processing file is included in the submission. The only thing that's a little wonky about this is when two Particles end up spawning inside one another, which causes one to usually get stuck inside the other while still registering as a collision.