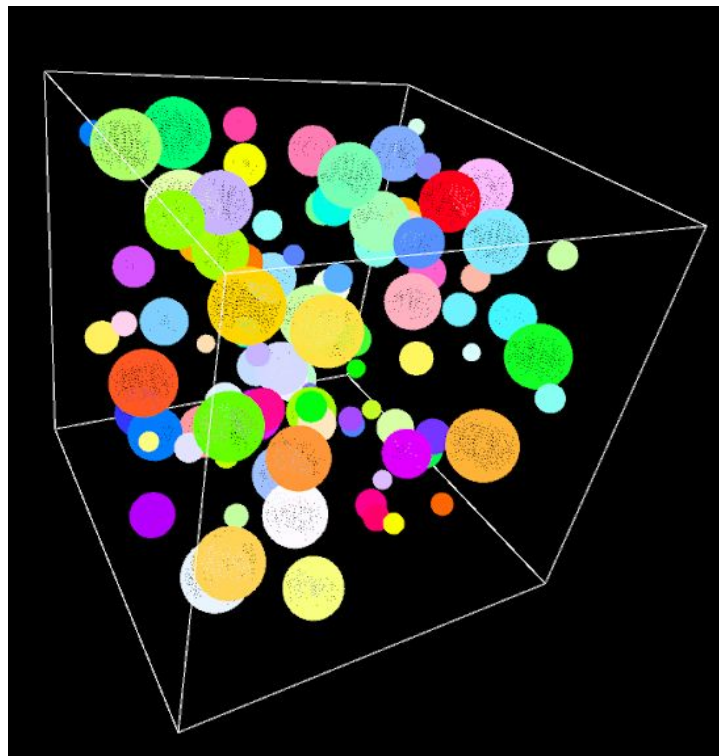


Three-Dimensional Collisions in Java

Trevor John Liggett



ABSTRACT:

This paper reviews the process I followed while implementing 3-Dimensional Collision Physics into computer code using the Java programming language. In this paper I have broken this process down into an introduction and the four phases of the computer program I created. These phases examine the physics and coding of collisions, as well as the problems I encountered along the way. Coding collisions is one of the hardest subjects I have delved into in high school, as all collisions must be calculated with no knowledge of any post-collision statistics. While the concepts I learned in AP Physics gave me the foundation to my understanding of these concepts, the collision formulas I learned were rendered useless, and I was forced to find answers elsewhere. This paper will describe these answers.

My coding is done in the Processing Developing Environment, created by Ben Fry and Casey Reas. Processing libraries make animation easier and I would strongly recommend similar libraries when working with advanced physics. Most of the physics used came from EuclideanSpace.com, a website developed by Martin John Baker devoted to Mathematics and Computing. Other works used are listed in the bibliography at the end of the paper.

TABLE OF CONTENTS

Page	#
Abstract	1
Table of Contents	2
Introduction Phase Zero: PVectors and Vector Motion	3
Phase One: Wall Collisions and the Matrix Stack	5
Phase Two: 3D Collision Physics with Two Objects	7
Phase Three: Multiple Objects	11
Phase Four: Adding Mass, Conserving Momentum, and an HSB color scheme	13
Glossary	15
Works Cited	16

Introduction Phase Zero: PVectors and Vector Motion

One of the most important aspects of bringing a third dimension into collisions is the inclusion of vectors. In either a two or three dimensional space, vectors have a direction and a magnitude. Because of this, vectors are our gateway to a third dimension, as we can simply add on a z-component and enter an entire new world of possibilities. In Processing, the **PVector** class was created to simplify the use of vectors in Java. PVectors store the x, y, and --depending on whether or not you are working with P3D-- z components as float types within the class. It contains methods to do pretty much anything you could want with a vector. Here are some of the functions I found helpful when working with collisions.

Figure 0.1

mag()	Calculates the magnitude of the vector.
add()	Adds x, y, and z components to a vector, one vector to another, or two independent vectors
sub()	Subtract x, y, and z components from a vector, one vector from another, or two independent vectors
mult()	Multiply a vector by a scalar
div()	Divide a vector by a scalar
dist()	Calculate the distance between two points
dot()	Calculate the dot product of two vectors
cross()	Calculate and return the cross product
normalize()	Normalize the vector to a length of one
limit()	Limit the magnitude of the vector

When mapping the motion of an object, tradition is to create both a location and a velocity vector.

During each tick of the program, the velocity vector is added to the location vector. If motion is to be altered, it is done through an acceleration or an impulse vector.

The class to the right is a primitive Ball class for my program. While there is no feature for multi-object collision, the Ball is designed to reverse direction if it reaches a certain distance from the origin. This distance is marked by a larger sphere boundary that is drawn, in

```
class Ball {
  PVector location;
  PVector velocity;
  float radius;
  color a;

  Ball(PVector loc) {
    location = loc;
    location.limit(0);
    velocity = new PVector(random(20)-10, random(20)-10, random(40)-20);
    a = color(random(255), random(255), random(255));
    radius = 50;
  }

  void move() {
    location.add(velocity);
    if (location.mag() + this.radius > width) {
      velocity.mult(-1);
    }
  }

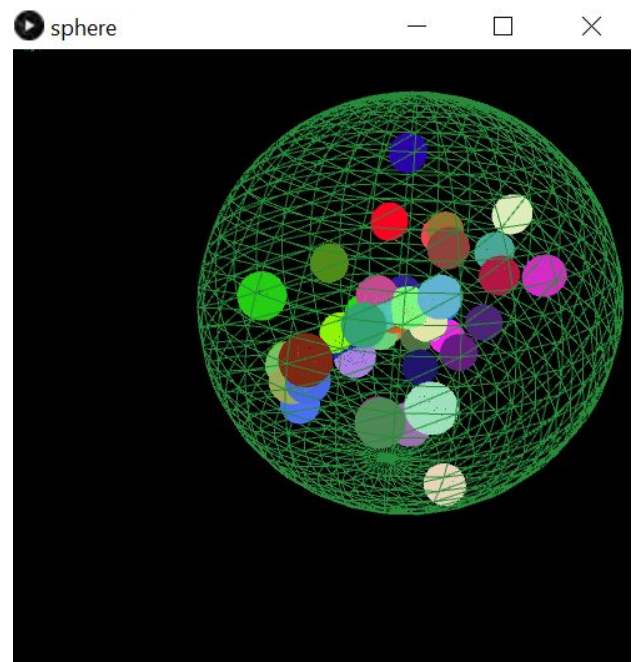
  void display() {
    pushMatrix();
    translate(location.x, location.y, location.z);
    noFill();
    stroke(a);
    sphere(radius);
    popMatrix();
  }
}
```

green, in Figure 0.2. With the code below, each ball is designed to “collide” with the outer boundary:

```
if (location.mag() + this.radius > width)
  velocity.mult(-1);
```

The distance of the edge of the Ball from the origin is calculated by adding the magnitude of the location vector and the radius of the Ball. If this sum is greater than the radius of the outer sphere, then the Ball’s velocity is multiplied by -1, reversing it’s direction and sending it back towards the origin.

While no Object-to-Object collision physics are in place in this program, this simulator, created in December 2016, was a key first step towards mastering PVectors and eventually collisions in Java. The Ball class created in this project is the foundation for what I would use in each of the programs I created for my final project.

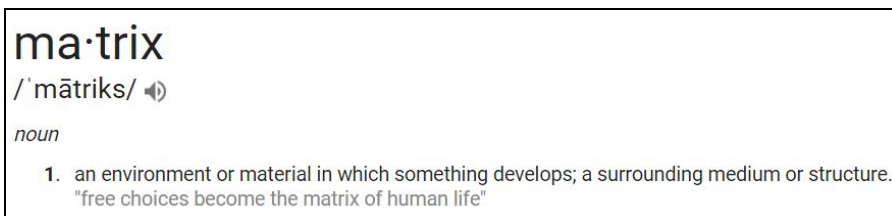


Phase One: Wall Collisions and the Matrix Stack

Bad news for collision junkies: Phase One is the setup of the Ball-Box system, and no Object-to-Object collisions will occur until Phase Two. With a basic knowledge of PVectors and the Matrix Stack, phases zero and one can be bypassed. However, these concepts are crucial to understanding my project. These phases are dense with code, and the truly “math-heavy” programming won’t occur until Phase Two. To understand how motion and graphics work in Processing, let’s explore the matrix stack.

The Matrix Stack

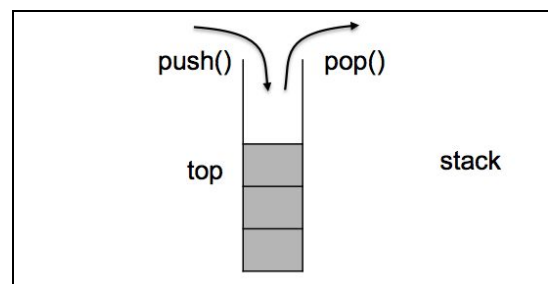
Backpedal for one second. What is a matrix?



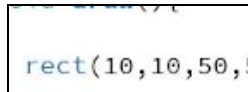
In the world of Processing Java, a **matrix** is the environment in which objects are being drawn. For example, to draw a rectangle at the point (10,10) in the coordinate plane, it can be drawn in two distinct ways: `rect(10,10,50,50);` or: `translate(10,10);`
`rect(0,0,50,50);`

The first, and simpler, way to draw the rectangle is to move it within its drawing call. The other way is to move the entire environment -- the matrix -- and then draw the rectangle. This may seem counterproductive, but when working with advanced translations and rotations with advanced objects, it can be easier to move the environment rather than the object. Thus, it is important to be able to change the matrix. But a matrix is just an object, which means there can be more than one matrix at a time. This is where the matrix stack comes into play.

A **stack** is a relatively simple data-type in Java, and some variation of a stack is a keystone to every coding language. Think of a stack like a stack of paper. When you put a piece of paper on top, it becomes the first piece that



you take off of the pile. This last-on, first-off concept is essential to a stack. When data is put onto the top of the stack in Java, it is being **pushed**. When data is taken off of the stack, it is being **popped**. The matrix stack in Processing is a stack that stores matrices for the program. When the current matrix is moved by the program, it can be a difficult process to move it back to its previous position. So before the current matrix is moved, a programmer must push it to the matrix stack. This saves the current configuration so it can be popped off the stack to return the matrix to its base position. These are the push and pop functions in Processing:



In each of the programs created for this project, the display method for the Ball class begins with pushMatrix() and ends with popMatrix().

Wall Collisions

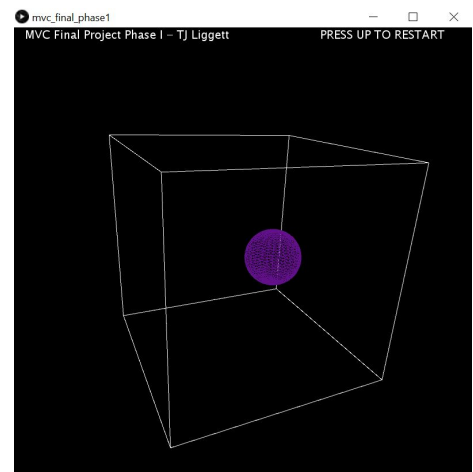
The wall collisions in Phase One were similar to the boundary collisions in Phase Zero, with one major difference: instead of an outer-sphere, the balls are now going to be contained inside of a box. When a ball hit the outer-sphere in Phase Zero, it rebounded in the exact same direction that it came. This makes for rather predictable and repetitive motion. With an exterior box, rebounding balls only change in a single dimension, making the motion a little more exciting. Most of the program is identical to the Phase Zero project, except the move() method identified above. This focuses on an individual dimension, and alters motion only in that direction. Here we access the primitive floats stored inside the PVector. We are able to edit these values as if they are their own individual data. While this is not the most exciting part of the project, it is important to set up the basic environment before we dive deep into the project.

```
void move() {
  location.add(velocity);

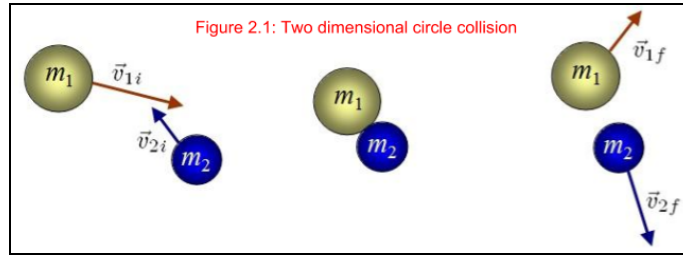
  if (abs(location.x) + radius > width/2)
    velocity.x *= -1;

  if (abs(location.y) + radius > width/2)
    velocity.y *= -1;

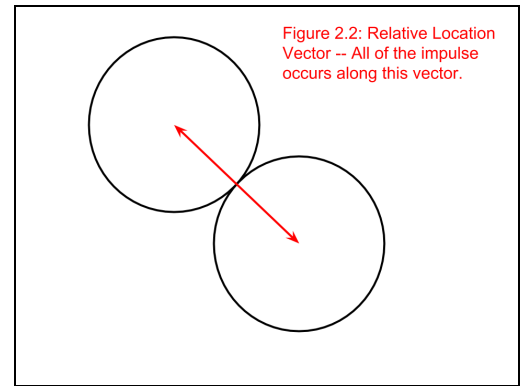
  if (abs(location.z) + radius > width/2)
    velocity.z *= -1;
}
```



Phase Two: 3D Collision Physics with Two Objects



As previously explained in phase zero, the computations behind three-dimensional collisions are connected to two-dimensional collisions through the use of vectors. In either case, vectors are vectors, with a little more variance in three-dimensions. To make the calculations simpler, all of the objects in the simulation are spherical, and while this is not realistic, it is certainly much simpler. When two spheres collide, an **impulse** is applied to each of the spheres. This impulse is transferred along a relative location vector (shown in Figure 2.2) we shall call the normal vector, the vector between the two midpoints of the spheres. This vector is normalized to a unit vector. The **Impulse-Momentum Theorem** states that the change in **momentum** of an object equals the impulse applied to it.



$$J = \Delta p = m\Delta v$$

Because our engine is setup to store location and velocity vectors, this equation is rearranged to get the change in velocity:

$$\Delta v = \frac{J}{m}$$

Working in three-dimensions, the equation must be “vectorized” in order to fit the program.

Taking what is already known about the transfer along the normal vector, this equation looks like this:

$$\begin{aligned} v_f - v_i &= -\frac{J}{m} \hat{n} \\ v_f &= v_i - \frac{J}{m} \hat{n} \end{aligned}$$

Remember in this equation that v and n are both vectors, so the direction of the impulse is shown through these vectors. Thanks to the website Euclidean Space, I found the formula for J , the impulse:

$$J = (1 + e) \frac{(v_a - v_b) \cdot \hat{n}}{\left(\frac{1}{m_a} + \frac{1}{m_b}\right)}$$

Attempting to implement this formula into code appears daunting, but by controlling initial variables we can see this simplified for the initial prototype. The first that can be simplified is the **coefficient of restitution**. This is the ratio of final to initial velocity of two objects when they collide.

$$\text{Coefficient of restitution } (e) = \frac{\text{Relative speed after collision}}{\text{Relative speed before collision}}$$

$$\text{The coefficient is related to energy by } e = \sqrt{\frac{KE_{(\text{after collision})}}{KE_{(\text{before collision})}}}$$

This ratio is normally somewhere between 0 and 1, as the objects will not gain energy from the collision. In a **perfectly elastic collision**, all energy is conserved, and thus the coefficient of restitution is 1. This is what we shall make the coefficient in our program. With no energy loss in the system, the program will continue forever and will not gradually slow down.

$$e = 1$$

We will also make the masses of all of the objects equal in phase two. The equation simplifies as follows:

$$\psi = \frac{J}{m} = (1 + 1) \frac{(v_a - v_b) \cdot \hat{n}}{(\frac{1}{m} + \frac{1}{m}) \cdot m}$$

$$\psi = (v_a - v_b) \cdot \hat{n}$$

$$v_{fa} = v_{ia} - \psi \hat{n}$$

$$v_{fb} = v_{ib} + \psi \hat{n}$$

We now have an equation that can be utilized in code. Here is how to implement it into code:

```
void collide(Ball one, Ball two) {
    PVector n = PVector.sub(one.location, two.location).normalize();
    println("n: " + n.mag());
    float psi = PVector.sub(one.velocity, two.velocity).dot(n);
    println("Om: " + psi);
    n.mult(psi);
    one.velocity.sub(n);
    two.velocity.add(n);
}
```

The PVector class is a major key, as the methods mult(), sub(), add(), mag(), normalize(), and dot() are all used. Implementing this into Java, I decided to develop an n and a psi vector, multiply the n vector by psi, and then add/subtract n from the velocity vectors. Because the same vector is being added to both velocities in the opposite direction, the total energy of the system stays the same, staying true to the elasticity coefficient. When building the program, it is not always obvious to tell based on relative velocities whether or not velocity is conserved. By finding the total kinetic energy of the system, there is an obvious indicator of whether or not your system is losing energy. If it is losing energy and the intent is a coefficient of restitution of 1, formulas are not being properly utilized.

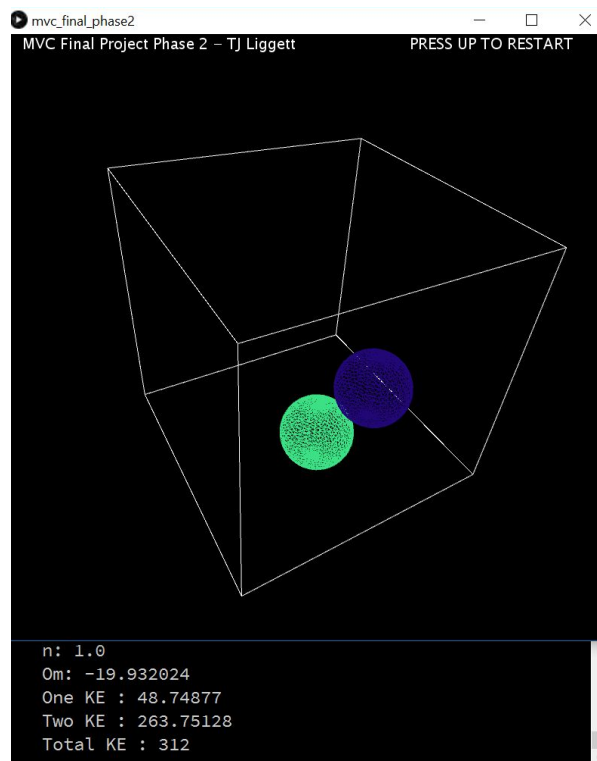
A problem that arose occurred with collision detection. Many times spheres in the simulator are still colliding during multiple ticks. In this scenario, the collision equation will occur twice. Thus, I created a **boolean** (a true/false value) that is marked based on whether the two spheres have already collided. If the spheres have just recently collided, then this collided boolean is marked as true, until the spheres have separated. This is implemented in code as such:

```
if ((ballOne.location.dist(ballTwo.location) < ballOne.radius + ballTwo.radius) && !collided) {
    collisions++;

    println();
    println("collision detected : " + (int)collisions);

    collided = true;
    collide(ballOne, ballTwo);
    println("One KE : " + pow(ballOne.velocity.mag(), 2)/2);
    println("Two KE : " + pow(ballTwo.velocity.mag(), 2)/2);
    println("Total KE : " + (int)(pow(ballOne.velocity.mag(), 2)/2 + pow(ballTwo.velocity.mag(), 2)/2));
} else if ((ballOne.location.dist(ballTwo.location) > ballOne.radius + ballTwo.radius)) {
    collided = false;
}
```

This phase was a major breakthrough in my program, as the spheres were finally colliding based on true physics.



Phase Three: Multiple Objects

Now that collisions are working with two objects, the next step is to expand into a larger amount of objects. Rather than creating new variables for each object, in computer science the smart thing to do is to create an **array** of objects that act in the same way. Then code is written to account for all of the objects, and an infinite-- or as many as the computer can handle-- amount of identical objects can be created. These objects have been created as such:

```
for(int i = 0; i < ballArray.length; i++)  
    ballArray[i] = new Ball();
```

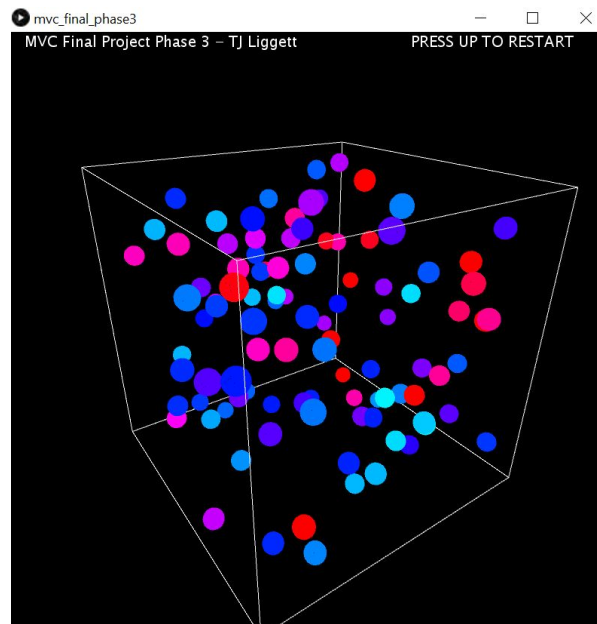
Rather than a Ball one and Ball two, a much larger array of balls, ballArray, now exists. The collisions for these balls are identical to the original program, except that we must tell the program which balls to use. However, the marking of these collisions becomes much more difficult as we add more objects. With two balls, there is only one possible collision occurring at any given time: ball 0 and ball 1. With three balls, there are now three. Five, ten. And so on. So along with creating an array of balls, we have to have somewhere and some way to store all of the collisions that have recently occurred in the system. To solve this problem, I created a Pair<E> class in Java. A **Pair** is a store of two objects who cannot be the same where order doesn't matter. Han and Chewie is the same as Chewie and Han. The Pair<E> class is shown in Figure. A **Set** contains only unique values, and each pair stores two values. An object's **hash code** is a unique value that differentiates it from other objects.

```
class Pair<E> {  
    private HashSet<E> items;  
  
    public Pair(E one, E two) {  
        items = new HashSet<E>();  
        items.add(one);  
        items.add(two);  
    }  
  
    public HashSet<E> getItems() {  
        return items;  
    }  
  
    public boolean equals(Object o) {  
        if (! (o instanceof Pair<?>)) {  
            return false;  
        }  
        Pair<E> p = (Pair<E>)o;  
        return p.getItems().equals(this.getItems());  
    }  
  
    public boolean equals(E one, E two) {  
        return items.contains(one) && items.contains(two);  
    }  
  
    public int hashCode() {  
        int h = 0;  
        Iterator<E> i = items.iterator();  
        while (i.hasNext()) {  
            E obj = i.next();  
            if (obj != null)  
                h += obj.hashCode();  
        }  
        return h;  
    }  
  
    public String toString() {  
        String output = "[";  
        for (E i : items) {  
            output += i + ",";  
        }  
        output = output.substring(0, output.length()-1) + "  
    }  
}
```

These pairs are used when the objects collide. The collision detection of multiple objects:

```
for (int j = i+1; j<ballArray.length; j++) {
    Pair<Integer> l = new Pair<Integer>(i, j);
    if ((ballArray[i].location.dist(ballArray[j].location) < ballArray[i].radius + ballArray[j].radius)) {
        if (!links.contains(l)) {
            collide(ballArray[i], ballArray[j]);
            links.add(l);
        }
    } else {
        links.remove(l);
    }
}
```

All of the active pairs are stored in a set that stores only unique values. That way if there are no repeated collisions. Rather than storing large objects in the pair, we simply store the integer index of each object in the ballArray. Before any collision detection occurs, a Pair<Integer> is created with the two objects' locations in the ballArray. Then, if the distance (dist() of the PVector class) is less than the objects' radii, and the Pair is not already in the list of active collisions (if its unique hash code, the sum of each of its objects hash codes, is not present in the list "links"), the collision will occur and the Pair will be added to the Set of Pairs, links in the code above. Now we can have a much larger amount of spheres in the collider with little change in the code!



Phase Four: Adding Mass, Conserving Momentum, and an HSB color scheme

With the base engine now, the next phase is to vary the masses of the spheres and keep energy conserved in the system. To make the equation simpler, the calculation is done for each ball using this equation for psi, where o is the foreign ball:

$$\psi = \frac{J}{m} = (1 + 1) \frac{(v - v_o) \cdot \hat{n}}{(\frac{1}{m} + \frac{1}{m_o}) \cdot m}$$

$$\psi = (2) \frac{(v - v_o) \cdot \hat{n}}{(1 + \frac{m}{m_o})}$$

```
PVector nA = PVector.sub(one.location, two.location).normalize();
float psiA = 2 * PVector.sub(one.velocity, two.velocity).dot(nA) / (1 + one.mass / two.mass);
nA.mult(psiA);

PVector nB = PVector.sub(two.location, one.location).normalize();
float psiB = 2 * PVector.sub(two.velocity, one.velocity).dot(nB) / (1 + two.mass / one.mass);
nB.mult(psiB);

one.velocity.sub(nA);
two.velocity.sub(nB);
```

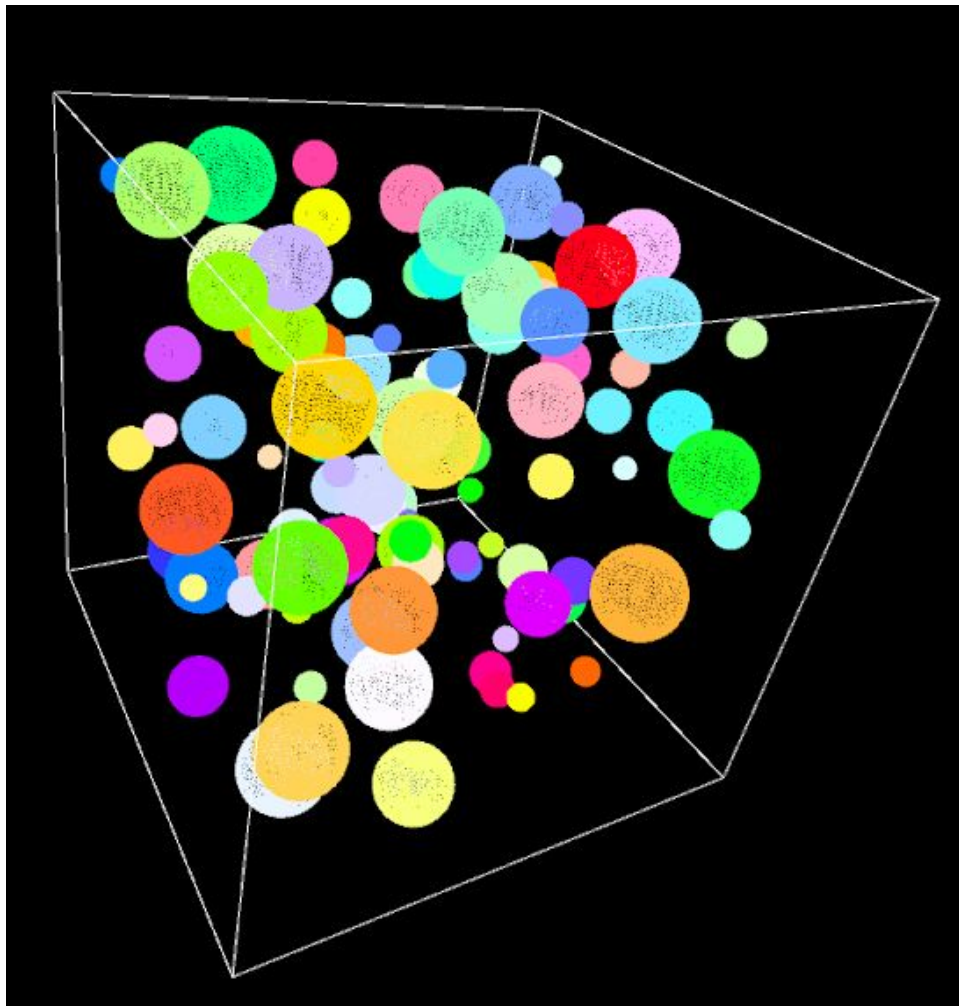
Another interesting twist in this phase was the introduction of the **HSB** color system: hue, saturation, and brightness. This is implemented into Processing with this: `colorMode(HSB, 360, 100, 100);` This is different from the **RGB** color system, where colors are given values of red, green, and blue. In the program, each Ball has a set hue and brightness, but the saturation is determined by the Kinetic Energy in the Ball:

$$\zeta = \frac{KE}{5}$$

$$\zeta = \frac{mv^2}{10}$$



With Phase Four, there is now a visual representation of energy transferred between different sized spheres in a frictionless cube. While this may not seem relatable to actual physics, this model accurately displays the physics of three-dimensional collisions in a visually appealing way. This model is also the gateway to many other possibilities in code. Any vision involving collision physics can be powered through the use of the physics discussed in this paper.



GLOSSARY

PVector- A class/datatype created to simplify the use of vectors in Processing. This class stores an x, y, and z float value and has many useful methods to complete vector calculations.

Matrix- An environment or material in which something develops; a surrounding medium or structure. In the world of Processing Java, this is the environment in which objects are being drawn.

Stack- A last-in, first-out data structure in Java, similar to a stack of paper.

push- To put/push an item on the top of a stack.

pop- To take the top item off the top of a stack.

impulse- The change in momentum of an object. This is the changing force in the program.

Impulse-Momentum Theorem- A theory that states the change in momentum of an object equals the impulse applied to it.

Momentum- The mass times the velocity of an object. When dealing with collisions, this is a major consideration.

coefficient of restitution- The ratio of final to initial velocity of two objects when they collide.

perfectly elastic collision- A collision in which no energy is lost; the total energy stays the same.

Boolean- A true/false datatype in coding.

Array- A store of multiple data values in Java that has a set size/length.

Pair- The data structure created for this project; stores two unique objects in which order does not matter.

Set- A data structure that stores only unique values, no duplicates.

hash code- The computers way of differentiating between objects of the same data structure.

HSB- A color description that includes hue, saturation, and brightness.

RGB- A color description that includes red, green, and blue.

WORKS CITED

- Baker, Martin John. "Physics - Collision in 3 Dimensions." Physics - Collision in 3 Dimensions - Martin Baker. Euclidean Space, 2015. Web. 22 May 2017.
- Deependra Singh, Nirapendra Singh, Deepak Sharma. "NCERT Solutions, CBSE Sample Paper, Latest Syllabus, NCERT Books, Last Year Question Papers and Many More ..." NCERT SOLUTIONS Video Solutions TEXT Books CBSE Sample Papers. NCERT Solutions, 2017. Web. 22 May 2017.
- Elert, Glenn. "Impulse & Momentum." Impulse & Momentum – The Physics Hypertextbook. The Physics Hypertextbook, 2017. Web. 22 May 2017.
- Fry, Ben, and Casey Reas. "Reference. Processing Was Designed to Be a Flexible Software Sketchbook." Processing. Processing Foundation, 22 May 2017. Web. 22 May 2017.
- Gupta, Lokesh. "Java Stack Implementation Using Array." HowToDoInJava. HowtoDoInJava.com, 30 Nov. 2015. Web. 22 May 2017.
- Merriam-Webster. "Matrix." Merriam-Webster. Merriam-Webster, 2017. Web. 22 May 2017.
- Physics, Mini. "UY1: Collisions." Mini Physics. Mini Physics, 14 Sept. 2015. Web. 22 May 2017.
- Wikipedia. "Coefficient of Restitution." Wikipedia. Wikimedia Foundation, 21 May 2017. Web. 22 May 2017.