Bates College

Senior Thesis

Mechanics Simulations Using Javascript

Author:
Peter Krieg

Advisor:
Gene Clough

Presented to The Department of Physics, Bates College
In Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science

Lewiston, Maine
December 1st, 2014

Declaration of Authorship

I, Peter Krieg, declare that this thesis titled, 'Mechanics Simulations Using Javascript' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this College.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

BATES COLLEGE

Abstract

Department of Physics

Bachelor of Sciences

Mechanics Simulations Using Javascript

by Peter Krieg

This thesis outlines the programming necessary to simulate various mechanical phenomena in physics. The simulations are programmed using JavaScript, and viewable in a web browser through the HTML5 canvas API. Please visit http://peterkrieg.com/thesis to view all of the simulations. Each chapter investigates a different topic of mechanics, with simulations for each. The first chaper investigates basic kinematics and aerodynamic drag through balls bouncing. The second chapter presents simulations for orbiting bodies, and investigates Kepler's 2nd law of planetary motion. The third chapter examines angular momentum, torque, and Newton's Second Law for rotation.

Acknowledgements

First and foremost, I would like to thank my advisor, Gene Clough. Gene, thank you for your guidance along the way: you helped formulate my ideas and encourage me throughout the semester. I appreciate your patience and dedication to working with me–I have always enjoyed discussing problems with you. I feel relieved walking into your office knowing that you will have a book pertaining any problem I'm having.

I would also like to thank Professor Hong Lin, who met with me multiple different times to discuss my thesis. Thank you, Professor Lin, for thoroughly explaining concepts with me and for letting me borrow some of your books. Additionally, thank you to Professor Mark Semon, for showing me past examples of theses, and outlining the formatting requirements for Bates. Thank you to the Ladd Library staff for helping me with research.

I would like to acknowledge Steven Gunn, who wrote the original LaTeX template that this thesis uses. All of the formatting and aesthetics of my thesis are thanks to his hard work, I only inputted my own content.

Thank you to my parents for encouraging me throughout the whole process, and for at least trying to understand what I was doing for my thesis. I would also like to thank my friends and classmates who supported my work.

Lastly I would like to thank the variety of online resources that helped me with coding challengs along the way. Stackoverflow was particularly useful with JavaScript and LaTeX problems I encountered.

Contents

Dedicated to my parents

Introduction

0.1 What is a Physics Simulation?

The purpose of this thesis is to present a series of physics simulations, each modeling a specific problem of physics as realistically as possible. These simulations differ from *animations*, which can be seen as predictable representations that always display the same visual. Animations are analagous to a movie script: no matter how many times you watch the movie, it will always end in the same way. Simulations, on the other hand, need to adapt to variable conditions, and be based partly on random processes. This brings up the topic of *dynamic* vs. *static* animation. Most of the physics simulations in this thesis will be dyamic because they present a unique viewing each time they are run, and can also involve user feedback which influences the outcome of the simulation.

Any simulation requires creating the illusion of motion. Almost every form of projected motion media uses frames to accomplish this. Researchers have shown that to make the simulation look realistic, it must be presented at a rate of around 60-100 frames per second. Anything far slower than this, and the human eye will detect the "choppiness" of the simulation. People can't detect anything much faster than 100 frames per second, so there is no need to project media faster than that, with the exception of slow-motion videography.

0.2 Methods of Producing a Simulation

The physics simulations in this thesis differ greatly from common animations. Movies and cartoons, for example, operate by displaying a series of images similar to one another, and displaying them as many frames per second to create the illusion of motion. My simulations, on the other hand, function by providing the *information* for each frame, and then providing the data for *how* the animation can be created. These instructions are passed onto the HTML5 canvas API, which creates the visual which can be seen in the web browser. Because physics simulations contain instructions instead of a series of images, the files of code take up far less space than a movie file would, for example. This is one primary advantage of coded simulations. Every simulation follows a similar set of steps, which can be simplified below:

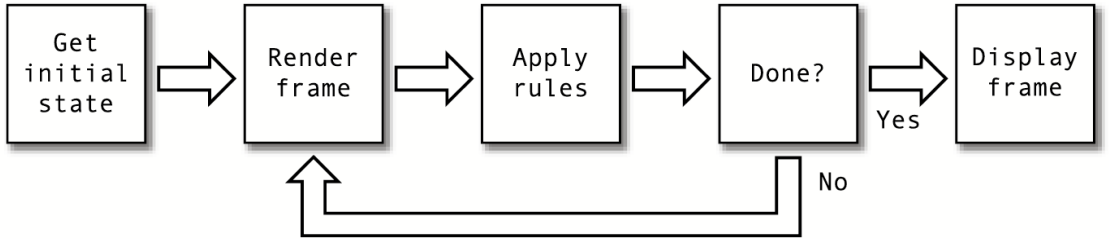

FIGURE 1: The frames of a general simulation

The canvas API gets the initial state of the simulation, which could the position of a ball, for example. Then, the frame is *rendered* by applying rules to the canvas element, and changing the initial state of the simulation. Once the rules have been applied, and all conditions are satisfied, the frame is rendered, and then displayed on the canvas element, to be seen in the web browser. The canvas is embedded into a web page with the ¡canvas¿ tag, like any other HTML tag. The positioning of objects in the canvas element is specified with a coordinate system, which uses pixels as its unit. Figure 2 shows the orientation of the canvas, which differs from the traditional cartesian coordinate system.

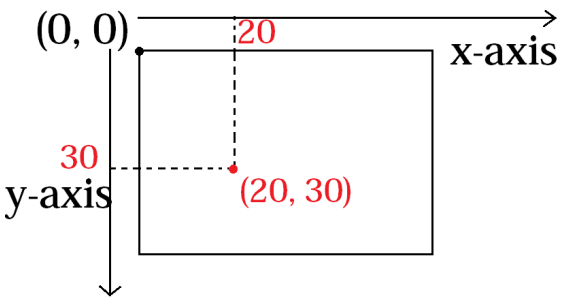

FIGURE 2: The canvas coordinate system. The canvas is displayed on the screen as an invisible white rectangle by default. A sample point of (20,30) is shown for clarification.

To produce any realistic simulation, the steps in figure 1 must be repeated multiple times per second. In fact, these steps must be repeated 60 times per second to achieve the desired 60 frames per second outlined in the previous section. Luckily, the canvas API is capable of running the instructions very quickly to make this simulation possible.

0.2.1 The Code

To program the simulations in this thesis, I chose to write the code in javascript. This scripting language is easy to view in any modern browser: therefore, all the simulations of this thesis can be viewed online. Javascript combines seamlessly with HTML5, which is why I mostly decided to use it for this thesis. The evolution of HTML (HyperText Markup Language) has progressed from simple web documents to complex web applications. For this thesis, every simulation utilizes the HTML5 <canvas>element, which has been used since around 2011. The HTML5 canvas API allows programmers to write javascript code that accesses the element and runs visual displays through a web browser The HTML needed to include a canvas can seen below:

```
1 <!doctype html>
2 <html>
3  <body>
4   <canvas id="canvas" width="500" height="500" >
5  </body>
6  <script>
7   var canvas = document.getElementById('canvas');
8   var context = canvas.getContext('2d');
9  </script>
```

```
10 </html>
```

Listing 1: The bare bones code necessary for an HTML document to include the canvas element

The above code displays the most basic HTML combined with javascript necessary to begin any simulation. Lines 7-8 are the only ones that actually contain javascript: this is the simple step necessary for the canvas API to recognize the HTML document. These two steps are necessary for any physics simulation. The step on line 7 initializes a JS variable and sets it equal to the canvas element on the web document object. The second step

All web browsers include some form of javascript interpreter: whenever the browser encounters a <script >element, it "passes" the code onto the JS interpreter. In listing 1, the HTML and JS code are written in the same document for clarity. While this is an acceptable practice, all future simulations will involve the HTML referencing to external JS documents to keep the contents separate. The appendices in this thesis have full code files from all the simulations.

While this thesis can contain code excerpts, figures, and screen-shots of various simulations, it obviously can't contain the flow of images itself. Therefore, I have put the entire thesis and its simulations on my personal website, which can be found at: http://www.peterkrieg.com/thesis. You can navigate by each chapter and view the simulations outlined in thesis.

Chapter 1

Some Basic Simulations

While the introduction outlined the computer programming necessary to produce simulations in general, this chapter will start to deal with the physics necessary to make simulations seem realistic. In this chapter, I will outline some examples of simulations with balls bouncing, and discussing the basic mechanics involved through the code.

1.1 Basic Ball Bouncing

A ball bouncing will show the basic kinematic equations, and how they are used in the javascript code. The following example displays a ball being dropped with an initial v_x, and bouncing off the walls and floor of the canvas element. The full code is shown below:

```
1  var canvas = document.getElementById(‘canvas’);
2  var context = canvas.getContext(‘2d’);
3
4  canvas.height = screen.height-200;
5  canvas.width = screen.width -100;
6
7  var radius = 20;
8  var color = ‘‘red";
9  var g = .1635; // acceleration due to gravity
10 var x = 40;  // initial horizontal position
11 var y = 40;  // initial vertical position
12 var vx = parseFloat(prompt(‘what is the initial horizontal speed of ball you
       would like?(recommended values of 1-20’));  // initial horizontal speed
13 var vy = 0;  // initial vertical speed
14
15 window.onload = init;
16
```

```
17 function init() {
18   setInterval(onEachStep, 1000/60); // 60 fps
19 };
20
21 function onEachStep() {
22   vy += g; // gravity increases the vertical speed
23   x += vx; // horizontal speed increases horizontal position
24   y += vy; // vertical speed increases vertical position
25
26   if (y > canvas.height - radius){ // if ball hits the ground
27     y = canvas.height - radius; // reposition it at the ground
28     vy *= -0.8; // then reverse and reduce its vertical speed
29   }
30   if (x > canvas.width - radius){ // if ball hits right wall
31     x = canvas.width - radius; // reposition it right at wall
32     vx *= -0.8;  // then reduce and reverse horizontal speed
33   }
34   if (x < radius){  // if ball hits left wall
35     x = radius;  // reposition it right at wall
36     vx *= -0.8  // then reverse and reduce horizontal speed
37   }
38   drawBall(); // draw the ball
39 };
40
41 function drawBall() {
42   with (context){
43     clearRect(0, 0, canvas.width, canvas.height);
44     fillStyle = color;
45     beginPath();
46     arc(x, y, radius, 0, 2*Math.PI, true);
47     closePath();
48     fill();
49   };
50 };
```

Listing 1.1: A basic ball bouncing simulation

This code functions by first setting up the canvas to be an appropriate size, on lines 1-5. Then, the simple variables of radius, color, initial positions/velocities, and acceleration are initialized. As mentioned in the introduction, the canvas HTML element defines positions in terms of pixels, with the top left corner of the canvas being the origin. Therefore, the ball is initialized to appear at (40, 40) which is near the top left corner for any computer screen. The value of g, the gravitational constant, is set to .1635 to accurately represent its value near Earth's surface, of 9.81 $\frac{m}{s^2}$. To understand why this value makes sense, it is necessary to understand the units of velocity on the HTML canvas. The position during the simulation is given in terms of pixels, which of course differs from the SI unit of meters. However, as long as g can be initialized to be 9.81 $\frac{px}{s^2}$, the simulation will still look physically accurate. This can be explained with the equation below:

$$9.81\ \frac{px}{s^2} = .1635\ \frac{\frac{px}{s}}{frame}\ \times\ \frac{60 frame}{s} \tag{1.1}$$

The value of g is calculated based on the fact that the simulation was run at 60 frames per second. Time is a central component of all physics, and for the simulations to behave realistically they must carefully take that into account.

The remainder of the code involves 3 functions that call one another to create the flow of the simulation. The first function, init ("initialize") is called when the browser window is loaded(line 15). This function simply delays the next function, onEachStep, by 16.66 ms, meaning that the function essentially runs 60 times per second, producing the desired 60 frames per second. Line 18 accomplishes this in a crude method: simulations later will involve more sophisticated techniques. The onEachStep function contains the instructions for each frame of the simulation. It involves multiple conditional if-statement loops that create the illusion that the ball bounces off of the walls and floor.

All 3 conditional loops involve a coefficient of restitution, or C_r. This is a mechanical property, representing how "bouncy" the ball is, and measures the ratio of the kinetic energy after and before the impact. This is derived below:

$$C_r = \sqrt{\frac{KE_f}{KE_i}} = \sqrt{\frac{\frac{1}{2}mv_f^2}{\frac{1}{2}mv_i^2}} = \frac{v_f}{v_i} \tag{1.2}$$

A C_r value of .8 was used for this simulation, which is comparable to that of a tennis ball[7]. The variable Cr represents this value in the code, and is simply multiplied by the velocity before impact, so the following equation results:

$$v_f = v_i * C_r \tag{1.3}$$

An essential part of any physics simulation involves *collision detection*. For the simple bouncing ball simulation, this is accomplished by conditional loops for if the ball's position exceeds the canvas constraints.

The last function of the program, drawBall, simply contains commands for the canvas API to draw. While these commands can be very complicated and intricate to create the exact visual aesthetic desired, the extent of these commands is not the purpose of this thesis. Basically, this function works by "erasing" the canvas of any previous graphics, and then creating a new visual with the arc() method.

The logic of the program can be summarized through the flow chart below:

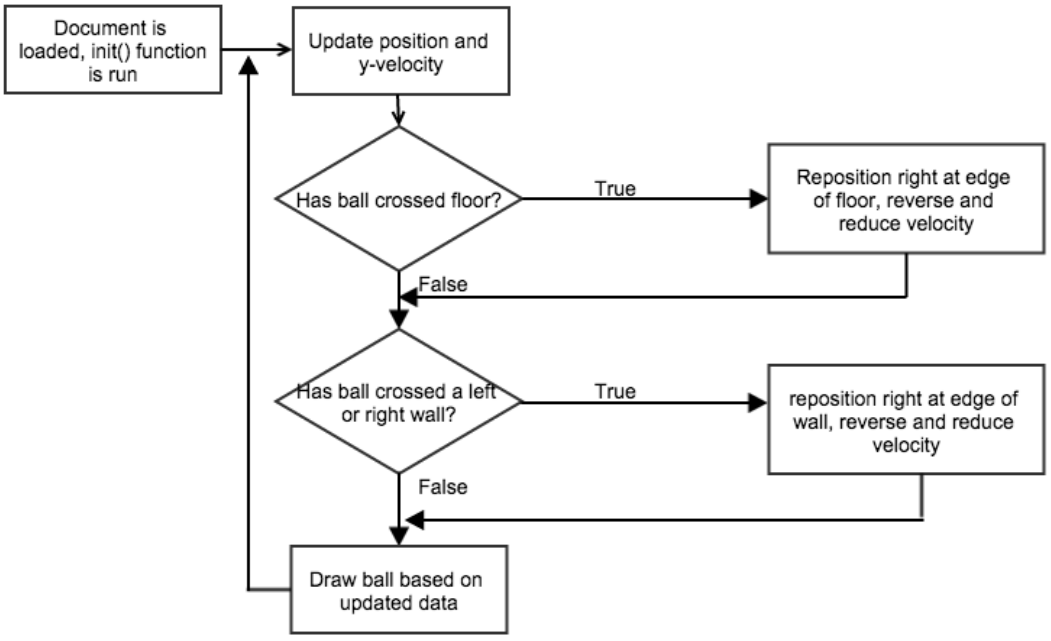

FIGURE 1.1: The logic flow chart of the basic bouncing ball simulation

1.2 More Advanced Ball Bouncing

While the previous example realistically incorporated the basic kinematic equations into account, it still fails to recognize important fundamentals of physics. The simulation in this chapter will still be a simple ball bouncing, but will take into account air resistance.

1.2.1 Background Physics

Drag is generally defined as the force on an object that resists its motion through a fluid. In the case of air resistance, the fluid is a gas, and therefore the process is called aerodynamic drag. Most of the drag force results as a response to the inertia of the fluid: the resistance it exerts to oppose being pushed aside. This can be expressed in the equation below:

$$f_{drag} = -\frac{1}{2} C_d \rho A v^2 \tag{1.4}$$

The equation involves a negative sign because the force of drag is always opposite the direction of motion. C_d is referred to as the drag coefficient, and is a dimensionless quantity that is used to model complex dependencies of shape, inclination, and flow conditions. While C_d is in general not an absolute constant for a given body shape, for the purpose of these simulations constant values were used. These values are typically determined experimentally: for example, the C_d of a sphere is approximately .47. In equation 1.4, ρ is the mass density of the fluid, in $\frac{kg}{m^3}$. Most of the simulations in this thesis occur in air, which has a density of 1.225 $\frac{kg}{m^3}$ (at sea level and 15 °C). Running the simulations in different fluids can be simulated by changing ρ to higher values (water, for example, would have ρ equal to 1000 $\frac{kg}{m^3}$). Lastly, A in equation 1.4 is the cross-sectional area of the object. A sphere, for example, would have a cross-sectional area of πr^2

The basic kinematic equations can also be used to make the simulations more physically realistic.

$$d = v_i t + \frac{1}{2} a t^2 \tag{1.5}$$

$$v_f = v_i + at \tag{1.6}$$

These equations are fundamental to any physical situation and can be used to make the ball bouncing example of the previous section more realistic

1.2.2 The Code

Using these basic mechanics equations, the previous ball bouncing example can be made more physically accurate. The code below shows a second simulation which incorporates air resistance:

```
var x = 40;
var y =40;
var vy = 0;
var ay = 0;
var m = 1;
var r = 20;
var rSI = r* 0.000230909;  // radius in SI, converting px to m
var C_r = .8;  // Coefficient of restitution (tennis ball would be .8)
var rho = 1.2;    // density of air would be 1.2, water would be 1000
var dt = 60/1000;  // Time Step
var C_d = 0.47; //Coefficient of drag for sphere
var A = Math.PI * rSI * rSI;
var color = 'red';

window.onload = init();

function init(){
  setInterval(onEachStep, 1000/60);
}

function onEachStep(){
  var fy = 0;
  fy += m * 9.81;   // weight force
  if (vy>=0){
    fy -= 1* 0.5 *rho * C_d *A *vy *vy;
  }
  else {
    fy += 1*0.5 *rho *C_d *A *vy *vy;
  }

  ay = fy / m;
  vy += ay * dt;
  y += vy;

  // simple collision detection for floor only
  if (y + r > canvas.height){
    vy *= -C_r;
    y = canvas.height - r;
  }
  drawBall();
}
```

Listing 1.2: More advanced ball bouncing simulation

To eliminate redundancy, the code doesn't show previous functions used, such as drawBall(). The code also doesn't show the basic steps to initialize any simulation with the canvas and context commands. The code is very similar to the simulation in the

previous section, except it incorporates air resistance. Essentially, this simulation uses more kinematic equations, by calculating the net force, acceleration, and velocity for each frame. First, the net vertical force is caculated, by combining the force of gravity $F_g = mg$ with the air drag from equation 1.4. This step involves a conditional loop for the cases of positive and negative velocity. Once the net force is calculated, the acceleration in the y-direction is found by using Newton's 2nd law of $F = ma$. From there, the velocity and vertical position of the ball are updated. Unlike the previous simulation, this example involves a variable dt, which is set to ∼16 ms for the same 60 frames per second.

This code involves interesting conversions between pixels and meters. Becuase the on screen simulation is presented eventually in terms of pixels, the physics equations must acknowledge this. The variable rSI on line 7 converts the radius of ball from pixels into meters. This is accomplished knowing the pixel density of the screen. This is commonly approximately 100 (dots per inch). The simulations were optimized for a macbook pro 15 inch model, which features 110 dpi. The calculation is shown below:

$$\frac{1m}{100cm}\,\frac{2.54cm}{1in}\,\frac{1in}{110px} \approx 0.00023091\frac{m}{px} \tag{1.7}$$

Once the conversion is made, the physics equations use the radius of the ball in terms of meters instead of pixels, which would give erroneous answers.

1.3 Multiple Balls Bouncing

So far, this chapter has dealt with a single object in motion. However, physics rarely involves just one body in motion. To demonstrate how more than one object can be displayed simultaneously, this section will show the case of multiple bouncing balls.

There is no new physics introduced in this section, but the coding concepts will be used repeatedly in later chapters of this thesis.

1.3.1 The Code

To generate more than one object, arrays can be used. The code below relies on arrays and object prototypes to create the effect:

```
var g = 0.1635;
var balls;
var numBalls = prompt('how many balls would you like to have bounce?');
var C_d = .8;

window.onload = init;

function init() {
  balls = []; // creates empty array
  for (var i=0; i<numBalls; i++){
    radius = Math.random()*20+5;
    var ball = new Ball();
    ball.x = 50;
    ball.y = 75;
    ball.radius =  radius;
    ball.vx = Math.random()*15;
    ball.vy = (Math.random()-0.5)*10;
    ball.color = getRandomColor();
    ball.draw(context);
    balls.push(ball);
  }
  setInterval(onEachStep, 1000/60); // 60 fps
};

function onEachStep() {
  context.clearRect(0, 0, canvas.width, canvas.height);
  for (var i=0; i<numBalls; i++){
    var ball = balls[i];
    ball.vy += g;

    if (ball.vx >0){ // while vx is positive, decrease to show friction/air drag
    ball.vx -= .001;
  } else{
    ball.vx === 0;// make sure ball stops moving appropriately
  }
    ball.x += ball.vx;
    ball.y += ball.vy;

    if (ball.y > canvas.height - ball.radius){
      ball.y = canvas.height - ball.radius;
      ball.vy *= -C_d;
    }
    if (ball.x + ball.radius > canvas.width){
      ball.x = canvas.width - ball.radius;
      ball.vx *= -C_d;
    }
    if (ball.x < ball.radius){
      ball.x = ball.radius;
      ball.vx *= -C_d;
    }
    ball.draw(context);
  }
};

function getRandomColor() {
    var letters = '0123456789ABCDEF'.split('');
```

```
57      var color = '#';
58      for (var i = 0; i < 6; i++ ) {
59          color += letters[Math.floor(Math.random() * 16)];
60      }
61      return color;
62  }
```

Listing 1.3: Multiple balls bouncing simulation

As with previous code listings, steps outlined in previous examples have been omitted to save space. This code differs mainly from previous examples because of its usage of prototypes, objects, and arrays. A separate javascript file, ball.js, contains the framework code for creating a ball. This will be used more in future chapters, so the code doesn't have to be repeated. This function is called a constructor function, becuase it allows other parts of code to reference the function when creating a new object. In the case of listing 1.3, an array holds an object for each different ball generated. The number of elements in the array is equal to the number of balls, which is selected by the user through the prompt() method on line 3. The object in each array element contains different properties for each ball: the radius, color, position, and velocities. For every frame of the simulation, a loop cycles through each element of the balls array, changing the properties of position and velocity, on lines 29 and 36-37. Exactly like in section 1, there is a conditional loop that controls the event of the ball colliding with a wall. The logic of this program can be visualized in the flow chart of figure 1.2.

Because of repeated for loops, this program involves a bit more complexity than the previous examples. However, the physics is very simple in this case. Future chapters will combine more complex physics with this complexity of coding to create more advanced simuations. These simulations can put a strain on a computer's performance: to simulate 20 balls bouncing, at 60 frames per second, 4,800 individual properties of objects need to be generated each second. Luckily, with the modern capabilities of computers, this isn't too difficult.

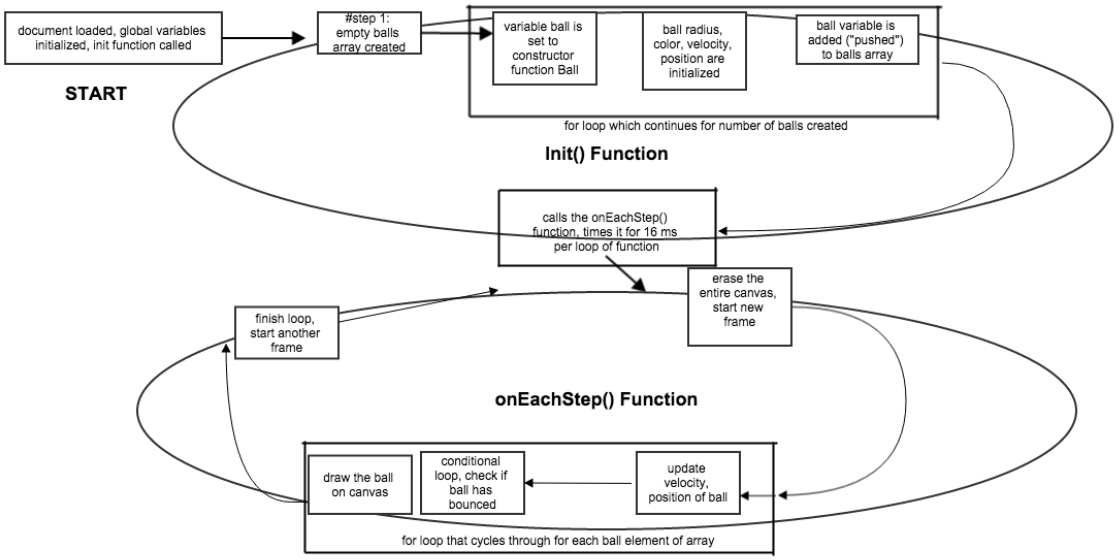

FIGURE 1.2: The logic flow chart of the balls bouncing simulation

This program involves each ball having a random color and radius, when the balls array is created in the init() function. The Math.rand() method is used for both of these, and is fundamental to the other simulations in this thesis. The random color generator function operates by creating a hex color by randomly assigning the 16 possible entries to each entry of the 6 character string. While completely unnecessary, this gives the program aesthetic appeal, and makes it easier to distinguish the balls.

Chapter 2

Simulating Orbits

In this chapter, a more advanced simulation of orbiting masses will be introduced. First, a simple orbit situation will be introduced, followed by more complex examples involving escape velocities. While the physics still isn't too advanced, the coding necessary is a bit more challenging.

2.1 Basic Orbit Path

The first simulation will deal with an example of a planet orbiting another much more massive planet.

2.1.1 Background Physics

This entire chapter is centered around Newton's law of universal gravitation:

$$F_g = G\frac{m_1 m_2}{r^2} \tag{2.1}$$

Where F_g is the magnitude of the force acting on either mass, G is the gravitational constant (SI units of 6.67 $\frac{Nm^2}{kg^2}$), m_1 is the mass of one object, m_2 is the mass of the

other object, and r is the radius separating the two masses. By Newton's 3rd law, there is an equal and opposite force exerted on each mass.

This equation can be used to describe the orbiting paths of planets. For simple cases when one planet orbits another, the variables m and M can be used. For a first example, we will assume M >>m, that is, one planet has a much greater mass than the other. Therefore, while each planet exerts an equal force on the other, the acceleration on the massive planet will be negligible. So, the smaller planet will orbit around the stationary planet, without attracting the larger planet enough to move it.

2.1.2 The Code

The full code is shown in the listing below:

```
1  var canvas = document.getElementById('canvas');
2  var context = canvas.getContext('2d');
3  var canvas_bg = document.getElementById('canvas_bg');
4  var context_bg = canvas_bg.getContext('2d');
5
6  var planet;
7  var sun;
8  var m = 1; // planet's mass
9  var M = 1000000; // heavy planet's mass
10 var G = 1;
11 var t0,dt;
12
13 window.onload = init;
14
15 function init() {
16         // create a stationary large planet
17         sun = new Ball(70,'orange',M);
18         sun.pos2D = new Vector2D(275,200);
19         sun.draw(context_bg);
20         // create a moving planet
21         planet = new Ball(10,'blue',m);
22         planet.pos2D = new Vector2D(200,50);
23         planet.draw(context);
24         // make the planet orbit the large planet
25         t0 = new Date().getTime();
26         animFrame();
27 };
28
29 function animFrame(){
30         animId = requestAnimationFrame(animFrame,canvas);
31         onTimer();
32 }
33 function onTimer(){
34         var t1 = new Date().getTime();
35         dt = 0.001*(t1-t0);
36         t0 = t1;
37         if (dt>0.1) {dt=0;};
38         move();
39 }
40 function move(){
41         moveObject(planet);
42         calcForce();
```

```
43          updateAccel();
44          updateVelo(planet);
45  }
46
47  function moveObject(obj){
48          obj.pos2D = obj.pos2D.addScaled(obj.velo2D,dt);
49          context.clearRect(0, 0, canvas.width, canvas.height);
50          obj.draw(context);
51  }
52  function calcForce(){
53          force = Forces.gravity(G,M,m,planet.pos2D.subtract(sun.pos2D));
54  }
55  function updateAccel(){
56          acc = force.multiply(1/m);
57  }
58  function updateVelo(obj){
59          obj.velo2D = obj.velo2D.addScaled(acc,dt);
60  }
```

Listing 2.1: Basic planet orbiting simulation

This program differs from previous ones used so far in that it uses two canvases instead of one. This is essential for having the large planet remain stationary and not being erased every frame. Instead, there can be a constant "background" canvas containing the stationary planet. The code begins by initializing the variables planet and sun, where sun simply refers to any large planet that has much more mass. The gravitational constant G is intialized as a formality just to a value of 1. G in this simulation isn't necessary, because the constant simply is used for unit conversion. This will become clear later in this section. When the web page is loaded, it calls the init function, just as in previous simulations. The init function creates the sun and planet as objects from the Ball constructor function, exactly as in chapter 1. Instead of having separate variables x and y in the previous examples, the position information can be stored into a property of each object, which is created using a different constructor function Vector2D.

The next function, animFrame, functions simply by initializing the javascript animation frame, and then calling the next function, onTimer. This next function creates a variable dt by converting the unit javascript operates in (ms) to SI units of s. It then passes the flow of the program onto the next function, move. This function involves

calling 4 functions, the first of which simply updates the position of the planet, erases the foreground canvas, and then draws the updated canvas. This step can be analyzed through a physics kinematics equation:

$$x(t + dt) = x(t) + v_x(t)dt \tag{2.2}$$

This is essentially analagous to Euler's method, by understanding that $v_x = \frac{dx}{dt}$.

$$x(t + dt) = x(t) + \frac{dx}{dt}(0)\,dt \tag{2.3}$$

By using constructor functions, with 2 different properties for the x and y position, the planet's location can be updated without updating variables and taking up more space. The location has to be updated for every frame, but so does the force, acceleration, and velocity. These next 3 steps are the remaining functions of the move function. The calcForce function updates the force of gravity acting on the planet, using equation 2.1. This equation calculates r by calculating the displacement vector between the two planets, and finding the magnitude of that vector. The updateAccel function simply takes the updated force vector and scales it by a certain "k" value which is represented by dividing by the mass. This is the step that incorporates Newton's 2nd law of $a = \frac{F}{m}$. The last function updates the velocity of the planet, similar to how the position was updated. Using Euler's method like before, we come to the following equation

$$v(t + dt) = v(t) + a(t)dt \tag{2.4}$$

Essentially, what makes this program more complicated is that it uses many other functions to accomplish the overall simulation. However, this method of programming makes future simulations easier–the same functions can be used, with changed variables.

The code listing below shows these "tool" functions that are used in future chapters of this thesis:

```
function Vector2D(x,y) {
        this.x = x;
        this.y = y;
}
Vector2D.prototype = {
        lengthSquared: function(){
                return this.x*this.x + this.y*this.y;
        },
        length: function(){
                return Math.sqrt(this.lengthSquared());
        },
        add: function(vec) {
                return new Vector2D(this.x + vec.x,this.y + vec.y);
        },
        subtract: function(vec) {
                return new Vector2D(this.x - vec.x,this.y - vec.y);
        },
        multiply: function(k) {
                return new Vector2D(k*this.x,k*this.y);
        },
        addScaled: function(vec,k) {
                return new Vector2D(this.x + k*vec.x, this.y + k*vec.y);
        },

        function Forces(){
}
Forces.gravity = function(G,m1,m2,r){
        return r.multiply(-G*m1*m2/(r.lengthSquared()*r.length()));
}
```

Listing 2.2: Various tools functions used for orbit simulation

2.2 Escape Velocity

The previous section tested situations where the planet orbited the sun continuously. However, if the speed is great enough, the orbiting body is capable of "escaping" from the larger planet's influence. The minimum speed necessary for this is called the escape velocity.

This can be derived by understanding conversation of energy. When an object leaves the surface of a planet, it will have an initial kinetic energy, and potential gravitational energy. This will equal the final potential energy, defined as a condition when the final kinetic and gravitational potential energy is 0. This relationship is shown in the equation below:

$$K_i + U_{g_i} = K_f + U_{g_f}$$

Knowing that the final kinetic and gravitational energy is 0, this equation becomes:

$$\frac{1}{2}mv_{esc}^2 - \frac{GMm}{r} = 0 + 0$$

Solving for v_{esc} yields the following:

$$v_{esc} = \sqrt{\frac{2GM}{r}} \tag{2.5}$$

Where G is the gravitational constant, M is the mass of the planet the object is escaping from, and r is the starting distance from the center of mass of the planet.

To test the physics behind the escape velocity, a slightly different scenario can be created with a different program. This simulation will have the object begin right at the surface of the larger planet, to emulate the process of "escaping" from the planet's gravity influence. To visualize this, a much larger canvas will be used, and some code changes will be utilized, seen below:

```
1 sun = new Ball(400,'orange',M);
2 sun.pos2D = new Vector2D(
3
4 planet = new Ball(10,'blue',m);
5 planet.pos2D = new Vector2D(500,2490);
6
7 planet.velo2D = new Vector2D(0, -80);
```

Listing 2.3: New conditions for escape velocity simulation

These changes made the larger planet look visually bigger, to simulate the effect of a massive planet. It also positioned the object to begin right on the surface of the larger planet (in this case, 410 pixels above the center of mass of the larger planet). To calculate the escape velocity for the situation above, equation 2.5 can be used, but understanding some key factors:

1. The escape velocity calculated will be in $\frac{px}{s}$ instead of SI unit $\frac{m}{s}$

2. The masses of each planet don't need to contain units, it can more just represent a ratio between the large and small planet. Therefore, each mass will be a unitless quantity, just used as a test of the escape velocity equation.

3. Similarly to #2, the gravitational constant G doesn't have to include units, since this test is only in a more theoretical sense, and doesn't use actual units of mass. However, using dimensional analysis, for the equation below to make sense, G could be viewed as having units of $\frac{px^3}{s^2}$.

Proceeding with these conditions in mind, the escape velocity for the simulation of code listing **??** can be calculated as shown below:

$$v_{esc} = \sqrt{\frac{2 * 1\frac{px^3}{s^2} * 1000000}{410px}} \approx 69.843\frac{px}{s} \tag{2.6}$$

Therefore, with the program simulation, any initial speed greater than this value will escape the gravitational influence of the larger planet. To test this, I used the following code to print out values of the velocity continuously:

```
1 var i =0;
2 i++;
3 if(i%15 ===0){console.log(planet.velo2D.length());}
```

Listing 2.4: Code for printing out values of speed

This code operates by printing out the value of the speed of object 4 times per second, understanding that i is incremented by 1 for each frame, and there are the usual 60 frames per second. The data is outputted through the console.log() method, which prints it onto the web browser. This data was then plotted for different initial speeds, and the results are shown below:

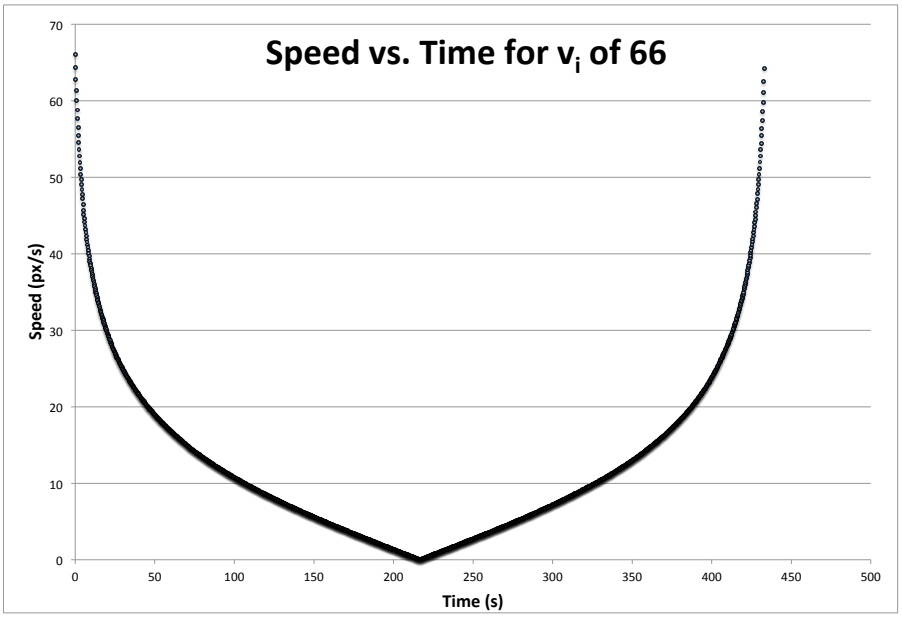

FIGURE 2.1: Speed vs. time for initial v_i of 66 $\frac{px}{s}$

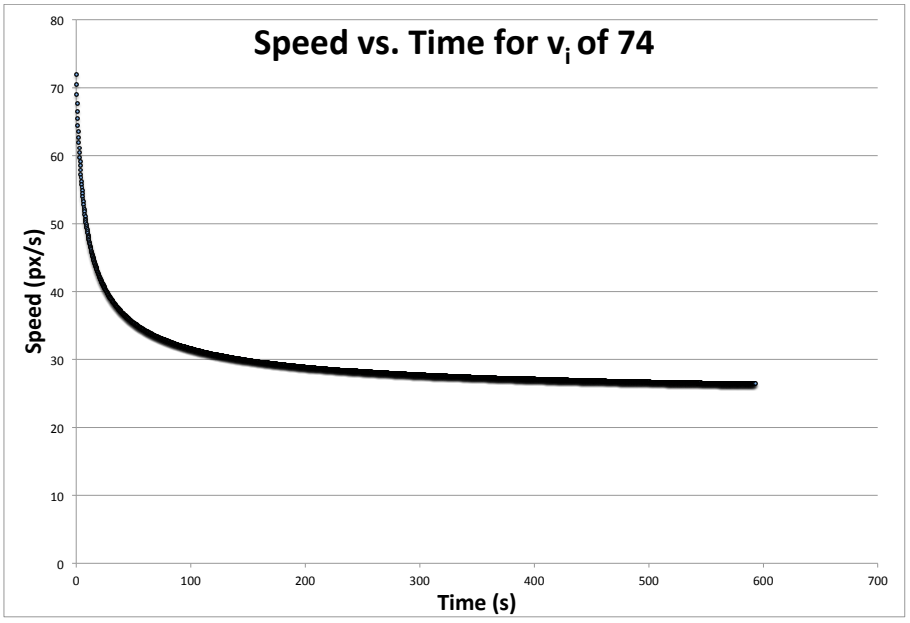

FIGURE 2.2: Speed vs. time for initial v_i of 74 $\frac{px}{s}$

These 3 figures show the stuff that happens with differet velocitias. need to talk about how near the escape velocity, intersting things happen. obvious results would occur far away from it.

These 3 figures show the speed vs. time for different intial velocities. Figure 2.1 shows an initial speed of 66, which isn't enough for the required 69.843 speed to leave

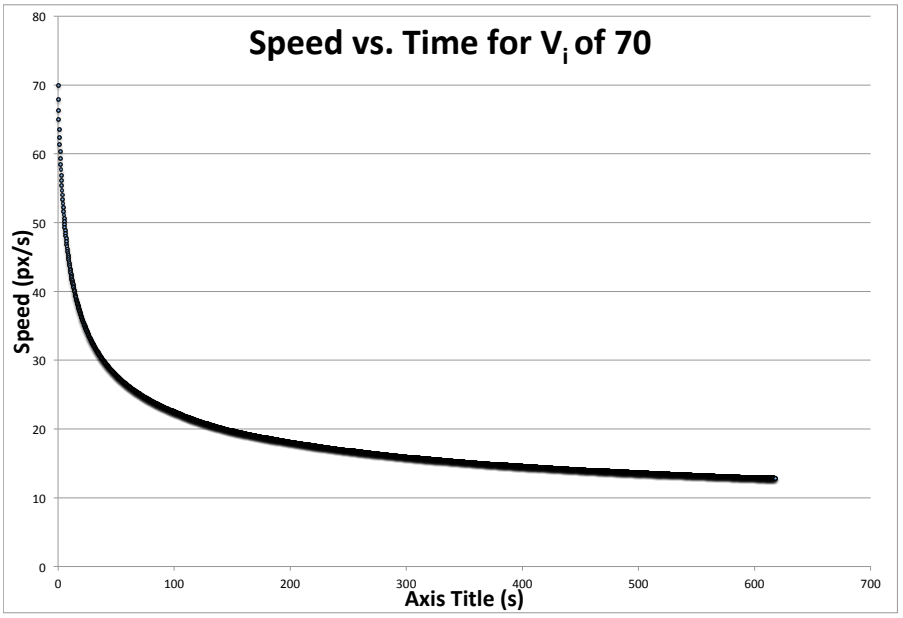

FIGURE 2.3: Speed vs. time for initial v_i of 70 $\frac{px}{s}$

the influence of the larger planet. The object rises far away from the planet, slows down, and then reaches a point where the speed is 0, and then reverse direction and accelerates back towards the planet. Figures 2.2 and 2.3 show initial speeds greater than that of the escape velocity, and the effect is clear: the speed tapers off eventually to an end velocity, as r approaches ∞ and the gravity force approaches 0. If the initial speed exactly equaled the escape velocity, in theory the final speed of the object would approach 0, and r approaches ∞. However, this simulation would take a very long time to do. All of these graphs were plotted over times ranging from 400 to 700 seconds, and since the speed was printed 4 times per second by the computer program, there were thousands of data points plotted overall.

2.3 Kepler's Laws

In the early 1600's Johannes Kepler proposed a series of laws that explained how planets orbit the sun. This was the support the scientific-based heliocentric model, which conflicted with the geocentric model before that. These 3 laws are shown below:

1. All planets move in ellpitical orbits with the Sun at one focus

2. The radius vector drawn from the Sun to a planet sweeps out equal areas in equal time intervals

3. The square of the orbital period of any planet is proportional to the cube of the semimajor axis of the elliptical orbit

The simulation in this section will help visualize law #2, using the orbit program already created in section 2.1 This law can be derived by understanding the situation of a planet orbiting the sun in an ellpitical. The sun is assumed to be much more massive so it doesn't move. At any instant along the path of orbit, the planet has a gravitational force pointing towards the sun, and it's velocity is tangential to this inward force. This can be visualized in figure 2.6.

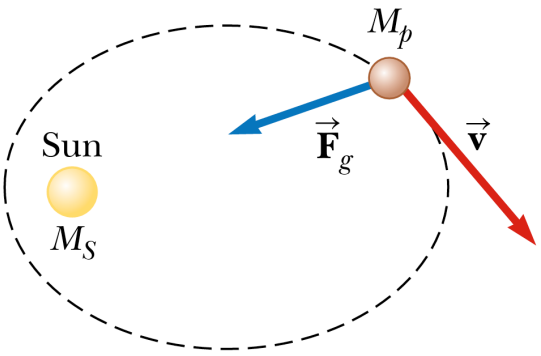

FIGURE 2.4: Basic model of planet orbiting a sun

The gravitational force is a central force that always points antiparallet to the radius vector $\vec{r}$. Knowing the radius vector and force on the planet at any point, the torque can be calculated, from the equation below:

$$\vec{\tau} = \vec{r} \times \vec{F}_g = \frac{d\vec{L}}{dt} \tag{2.7}$$

Because the radius and force vector are always antiparallel to one another, the torque, and therefore, the change in angular momentum will equal 0. In other words, $\vec{L}$ will remain constant. Knowing that $\vec{p} = M_p \times \vec{v}$, the following can be derived:

$$\vec{L} = \vec{r} \times \vec{P} = M_p \vec{r} \times \vec{v}$$

However, since the angle between $\vec{r}$ and $\vec{v}$ is always 90°, the equation above can expressed as:

$$L = M_p \left| \vec{r} \times \vec{v} \right| \tag{2.8}$$

This equation can be related to figure 2.5, which shows the relationship between $\vec{r}$ and $d\vec{r}$.

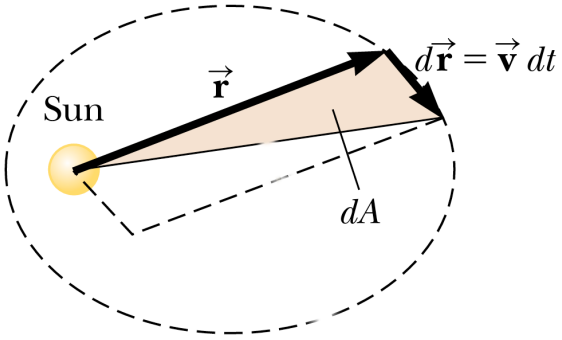

FIGURE 2.5: Relationship between $\vec{r}$ and $d\vec{r}$

$\vec{r} \times d\vec{r}$ equals the area of the parallelogram in figure 2.5. As $dt \to 0$, the area dA equals $\frac{1}{2}$ the area of this same parallelogram. Using this relationships yields the following:

$$dA = \frac{1}{2} |\vec{r} \times d\vec{r}| = \frac{1}{2} \left| \vec{r} \times \vec{v} dt \right| = \frac{1}{2} \left| \vec{r} \times \vec{v} \right| dt$$

Rearranging equation to solve for $|\vec{r} \times \vec{v}|$, and substituting this into the above expression yields the following:

$$dA = \frac{1}{2}\left(\frac{L}{M_p}\right)dt$$

Finally, dividing both sides by dt yields the final equation:

$$\frac{dA}{dt} = \frac{1}{2}\left(\frac{L}{M_p}\right) \tag{2.9}$$

Since L and M_p are constants, this equation shows that the rate of change in area is constant. To visualize this, slight adjustments were made to the simulation of section 1. The code changes are shown below:

```
1  if(i<960){
2          if(i%30===0){
3          context.strokeStyle = 'white';
4          context.moveTo(planet.x, planet.y);
5          context.lineTo(sun.x, sun.y);
6          context.stroke();
7          var dr = Vector2D.distance(planet.pos2D, planet.oldpos2D);
8          var r =  Vector2D.distance(planet.pos2D, sun.pos2D);
9          console.log('dA is equal to:  %f', .5*r*dr);
10         planet.oldpos2D=planet.pos2D;
11         }
```

Listing 2.5: Code for printing out values of speed

This code creates a condition where ever .5 seconds, a line is drawn between the position of the sun and the position of the planet. This visualizes the display of breaking up the orbit path into different area segments, which should all be equal area. Because the simulation occurs at a consistent rate of 60 frames per second, the lines could be drawn at a consistent rate over time. The code also calculates a variable r, which is the magnitude of the displacement vector between the two positional vectors of the planet and sun. The vector dr is also calculated by comparing the positional vectors of the planet between two different times. With these two variables, the program performs a rough calculation of dA, by understanding it is approximately $\frac{1}{2}$ the area of the parallelogram. The areas were the same within a reasonable amount of uncertainty and accuracy possible with the javascript program. The smallest "dt" possible in this program is 17

ms due to the limitations of the animation method of javascript. However, if dt could be made to approach 0, the calculations of dA would likely be closer to one another.

A screenshot of the simulation is shown below for reference:

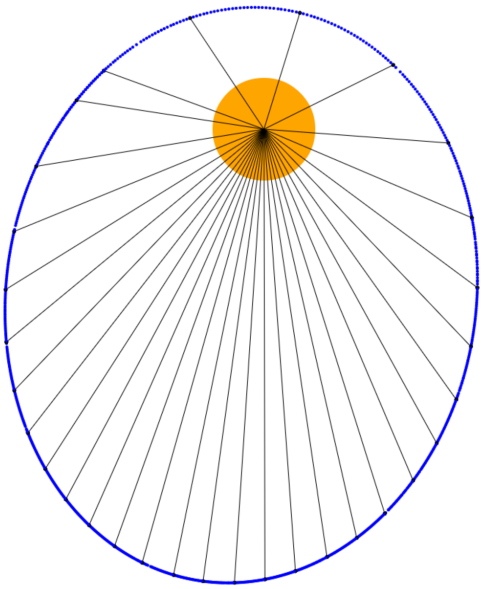

FIGURE 2.6: Screenshot of kepler law test simulation

The screenshot helps visually understand Kepler's 2nd law by separating areas of the same change in time by lines.

Chapter 3

Rigid Body Motion

Previous chapters have disregarded rotational motion of solid bodies. This chapters examines the mechanics of rigid body rotations and collisions through angular momentum fixed axis rotation. Simulations will involve rigid body collisions.

3.1 Angular Momentum and Torque

Previous sections in this thesis have simply involved translational motion. The theorem of rigid body motion states that the displacement of any rigid body can be decomposed into two independent motions: the translation of the center of mass, and the rotation about the center of mass. A rigid body in general is defined as an object that maintains its shape and size when a force is applied to it. In reality, all objects experience some level of deformation, however for the purpose of these simulations it is safe to ignore this.

To rotate a rigid body about an axis, a force must be applied to create a moment of torque, given by the equation below:

$$\vec{\tau} = \vec{r} \times \vec{F} \tag{3.1}$$

Where $\vec{r}$ is the vector from center of rotation to the point of application of force. If the force applied has a line of action that intersects the center of mass of the object, no torque is produced. To simplify the simulations involving toruqe, all objects are polygons with an assumed uniform density. Therefore, the center of mass would always be the geometric center of the polygon. The center of mass is essential for simulations becuase it is the reference point that allows the simulations to involve both translational and rotational motion.

While moment of torque represents resistance to angular motion, moment of inertia represents resistance to angular *acceleration*. For a continuous distribution of mass like a rigid body, the moment of intertia is defined by the following:

$$I = \int r^2 dm \tag{3.2}$$

Lastly, while linear momentum is related to translational motion, angular momentum is related to rotational motion. Angular momentum is defined by: $\vec{L} = \vec{r} \times \vec{p}$. A rigid body can be interpreted as a collection of particles all rotating with the same angular velocity ω, which allows for the following equation:

$$L = \sum m_i r_i^2 \omega \tag{3.3}$$

Using the definition of moment of inertia, the above equation can be written as:

$$\vec{L} = I\vec{\omega} \tag{3.4}$$

Understanding these fundamental equations is the background for Newton's Second Law for rotational motion, which is essential for rigid body simulations. Any rigid body can be visualized as a collection of infinitely small particles of mass dm, as shown below:

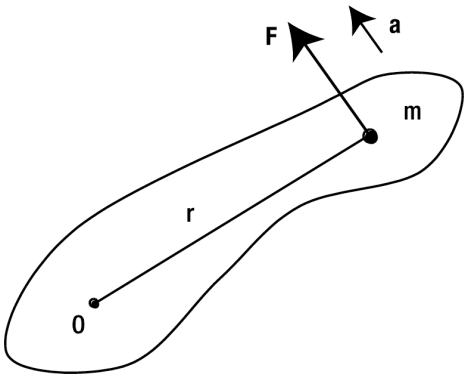

FIGURE 3.1: Diagram of rigid body with center of mass O

Understanding the equation $a = r\alpha$ and substituting into Newton's 2nd law yields the following:

$$F = mr\alpha \tag{3.5}$$

Manipulating the scalar equation for torque of $T = Fr$ and combining with the above equation yields:

$$T = \sum mr^2\alpha \tag{3.6}$$

Again, understanding the equation for moment of inertia, and subsituting back for vectors yields the following:

$$\vec{T} = I\vec{\alpha} \tag{3.7}$$

This formula is analagous to the common Newton's 2nd law of $F = ma$ for rotational motion, instead of translational motion. From this equation, the angular acceleration of any rigid body can be calculated from the torque, which is essential for creating simulations of rigid body rotation.

3.1.1 Rigid Body Experiencing Torque

To create a basic simulation of an object experiencing torque, it is crucial to understand how to incorporate rotation. This is done with the following code:

```
rotate: function(angle){
                return new Vector2D(this.x*Math.cos(angle)-this.y*Math.sin(angle)
    ,this.x*Math.sin(angle)+this.y*Math.cos(angle));
        }
```

Listing 3.1: Code to rotate an object

The rotate method of the Vector2D object returns the updated vector based on the value of the angle argument when the method is called. The rotate method can be derived using trigonometry, and by understanding the unique difference of the canvas coordinate system described in the introduction. Essentially, for an angle of rotation θ, the resultant vector would be $(xcos\theta - ysin\theta)\imath + (xsin\theta + ycos\theta)\jmath$.

```
var rigidBody;
var acc, force;
var alp, torque;
var t0, dt;
var animId;
var kLin = 0.05; // linear damping factor
var kAng = .5; // // angular damping factor

window.onload = init;

function init() {
        var v1 = new Vector2D(-100,100);
        var v2 = new Vector2D(100,100);
        var v3 = new Vector2D(100,-100);
        var v4 = new Vector2D(-100,-100);
        var vertices = new Array(v1,v2,v3,v4);
        rigidBody = new PolygonRB(vertices);
        rigidBody.mass = 1;
        rigidBody.im = 5;
        rigidBody.pos2D = new Vector2D(500,200);
        rigidBody.velo2D = new Vector2D(30, 0);
        rigidBody.angVelo = 0;
        rigidBody.draw(context);
        t0 = new Date().getTime();
        animFrame();
};

function animFrame(){
        animId = requestAnimationFrame(animFrame,canvas);
        onTimer();
}
function onTimer(){
        var t1 = new Date().getTime();
        dt = 0.001*(t1-t0);
        t0 = t1;
        if (dt>0.2) {dt=0;};
        move();
}
function move(){
        moveObject(rigidBody);
```

```
41          calcForce(rigidBody);
42          updateAccel(rigidBody);
43          updateVelo(rigidBody);
44  }
45  function moveObject(obj){
46          obj.pos2D = obj.pos2D.addScaled(obj.velo2D,dt);
47          obj.rotation = obj.angVelo*dt;
48          context.clearRect(0, 0, canvas.width, canvas.height);
49          obj.draw(context);
50  }
51  function calcForce(obj){
52          force = Forces.zeroForce();
53          force = force.addScaled(obj.velo2D,-kLin); // linear damping
54          torque = 1;
55          torque += -kAng*obj.angVelo; // angular damping
56  }
57  function updateAccel(obj){
58          acc = force.multiply(1/obj.mass);
59          alp = torque/obj.im;
60  }
61  function updateVelo(obj){
62          obj.velo2D = obj.velo2D.addScaled(acc,dt);
63          obj.angVelo += alp*dt;
64  }
```

Listing 3.2: Code for angular rotation simulation

Bibliography

[1] "Air Friction." Air Friction: Viscous Resistance. Georgia State University. Web. 2 Nov. 2014. ¡http://hyperphysics.phy-astr.gsu.edu/hbase/airfri.html¿.

[2] Flanagan, David. JavaScript: The Definitive Guide. 6th ed. Beijing: O'Reilly, 2011. Print.

[3] Hawkes, Rob. Foundation HTML5 Canvas. New York: Friends of ED :, 2011. Print. Lamberta, Billy, and Keith Peters. Foundation HTML5 Animation with Javascript. New York: Friends of Ed :, 2011. Print.

[4] Kleppner, Daniel, and Robert J. Kolenkow. An Introduction to Mechanics. New York: McGraw-Hill, 1973. Print.

[5] Ramtal, Dev, and Adrian Dobre. Physics for JavaScript Games, Animation, and Simulations: With HTML5 Canvas. Apress. Print.

[6] Rauschmayer, Axel. Speaking JavaScript. Cambridge: O'Reilly, 2011. Print.

[7] Roux, A., & Dickerson, J.(2007). ISB Journal of Physics *Coefficient of Restitution of a Tennis Ball.* Retrieved November 10, 2014, from http://www.isb.ac.th/HS/JoP/vol1/Papers/Tennis.

[8] Serway, Raymond A., and John W. Jewett. Physics For Scientists and Engineers with Modern Physics. 9th ed. Boston, MA: Brooks/Cole, Cengage Learning, 2014. Print.

[9] Young, Hugh D., and Roger A. Freedman. University Physic with Modern Physics. 13th ed. Pearson, 2011. Print.