Documentation for Programming Summative – Patrick Coglan (sjtj97)

Source of Sketch and Licencing

The sketch which I used was sourced from OpenProcessing (https://www.openprocessing.org/sketch/635788).

I am free to share and adapt the sourced code under the **Creative Commons ShareAlike 3.0 Unported (CC BY-SA 3.0)** licence.

Note

From the date of sourcing the code from OpenProcessing the code may have changed due to further development from the original creator. The code which I sourced has been zipped and matches the exact code which was obtained at the time and can be found in the ‘Source Code’.

Explanation of Example

The example I have chosen is designed to simulate solar scale gravity. By using Newtons law of gravitation the example is able to simulate the forces that particles feel toward other objects (attractors), with my implementation you the user is able to control many more properties of each individual particle and attractor, giving them the freedom to replicate many different objects that can be found in the universe. Not only have I added in a variety of properties, I have added the ability to create particles on a mouse drag of which have the direction with respect to the mouse drag direction and also the velocity scaled to the magnitude of the drag line, this gives the user much more freedom in how particles act within the scene. Furthermore, this much customisation allows the player to simulate circular motion.

Explanation of Code

There are four components that make up my implementation of the code.
The files that my project contains is:

- index.html – The homepage of the website
- style.css – The style (properties) of the index.html document
- sketch.js – The game (gravity simulation)
- p5.dom.js – The p5 DOM library

The HTML Document (index.html)

```
<!DOCTYPE html>

<html lang = 'en'>
    <head>
        <!-- Importing p5js libraries (connection needed)
-->
        <script src = "https://cdnjs.cloudflare.com/ajax/l
ibs/p5.js/0.7.2/p5.js"></script>
        <script src = "p5.dom.js"></script>
```

```html

        <!-- Importing modified sketch  -->
        <script src = "sketch.js"></script>

        <!-- Linking a stylesheet -->
        <link rel = "stylesheet" type = "text/css" href = "style.css">

        <!-- The title of the page -->
        <title>A p5js Gravity Simulation</title>
    </head>

    <body>
            <div class = "head" id = "header">
                <h1 id = "title">A p5js Gravity Simulation </h1>
                <h2 id = "subtitle">Modified by Patrick Coglan (sjtj97)</h2>
                <div class = "PlayButton" id = "play" onclick = "transitionToGame()"><h1 id = "play">Play Game</h1></div>
            </div>
    </body>
</html>
```

The `head` tags are used to reference all important components that the webpage requires to function. So there is a reference to, two p5 libraries, the game itself and finally the stylesheet. The title is also specified here.

The `body` tags are used as a section which content is displayed to the user. My `body` section contains a master `div` element which houses the information displayed to the user. Within this master `div` are two title tags (`h1` and `h2`) and a nested `div`. This nested `div` acts as a button so the user can press it to play the game. When the user presses on the `div` a function named `transitionToGame()` is called from the sketch.js file.

The CSS Stylesheet (style.css)

```
html, body{
    padding: 0px;
    margin: 0px;
    background-color: black;
}

.head {
    position: fixed;
    font-family: monospace;
    text-align: center;
    background: #4d0000;
    color: white;
    height: 100%;
    width: 100%;
    cursor: default;

```

```css
}

.head #title {
    font-size: 100px;
    font-size: 10vw;
}

.head #subtitle {
    font-size: 50px;
    font-size: 3vw;
}

.PlayButton {
    position: fixed;
    bottom: 0;
    background-color: black;
    width: 100%;
}
.PlayButton:hover {
    background-color: #262626;
}

.PlayButton:hover #play {
    cursor: pointer;
    transform: scale(1.5);
    transition: transform 0.5s;
}
```

```
.PlayButton #play {
    transform: scale(1);
    transition: transform 0.5s;
}
```

The block of CSS below is used to set all the padding and margine values of the `html` and `body` tags to zero and also the background colour property to black. Padding and margins are removed so that the `div` elements can fit the full width and height of the browser.

```
html, body{
    padding: 0px;
    margin: 0px;
    background-color: black;
}
```

The block of CSS below is applied to any element with the class attribute set to “head”. So this styling is applied to the master `div` element. I have set the position of this element to fixed so that the element is displayed on the screen at all times, this also removed the scroll bars at the bottom and right of the page when the `div` element was fullscreen. All text and background properties were set here also. The height and width properties are both set to 100% to keep the div fullscreen at all times, even after a window resize. Finally the cursor property is set to default so that there is no change in cursor when hovering over the text on the `div`.

```css
.head {
    position: fixed;
    font-family: monospace;
    text-align: center;
    background: #4d0000;
    color: white;
    height: 100%;
    width: 100%;
    cursor: default;
}
```

The two block of CSS below are used to define the font size of the text within the head class and of a specific element ID. In each block I have set a default font-size property so that if the other property for some reason fails, I have a fixed failsafe. The other properties override the font-sze by setting it to a value with **vw**, by doing this I am able to force the text to rescale whenever the browser window is resized by the user, this ensures that as the browser gets smaller, so does the text. This ensures that the text is always visible to the user and that the text is not too large for the window size.

```css
.head #title {
    font-size: 100px;
    font-size: 10vw;
}

.head #subtitle {
    font-size: 50px;
    font-size: 3vw;
}
```

Lastly, the below CSS is used to style the button at the bottom of the page. It is responsible for settings the background colour, hover colour, enlarging the text when hovering, setting the `div` position to the bottom etc. As before, the position of the `div` with the PlayButton class is fixed so no scrolling can occur.

```
.PlayButton {
    position: fixed;
    bottom: 0;
    background-color: black;
    width: 100%;
}
.PlayButton:hover {
    background-color: #262626;
}

.PlayButton:hover #play {
    cursor: pointer;
    transform: scale(1.5);
    transition: transform 0.5s;
}

.PlayButton #play {
    transform: scale(1);
    transition: transform 0.5s;
}
```

The Javascript Simulation (sketch.js)

In this section I will explain how my JavaScript functions. I will do it class by class and function by function.

The Particle Class

```
// Used to create new particles objects
class Particle{
    constructor(x, y, vx, vy, velocityVector, sliderValues
){
        this.Color = sliderValues[0] || 0.8;
        this.Size = sliderValues[1] || 0.8;
        this.Mass = 1;
        this.Friction = 0.1 * sliderValues[4] || 0;
        this.Location = createVector(x, y) || createVector
(window.innerWidth/2, window.innerHeight/2);
        this.Velocity = velocityVector.mult(sliderValues[3
]) || createVector(0,0);
        this.Acceleration = createVector(vx, vy) || create
Vector(0,0);
        if(this.Friction == 0.1){this.Friction=1;}
        this.createParticleImage();
    }
    // Used to update the particles velocity and accelerat
ion and take in account friction (if there is any)
    update(){
        this.Velocity = this.Velocity.add(this.Acceleratio
n);
        this.Velocity = this.Velocity.mult(1-this.Friction
);
        this.Location = this.Location.add(this.Velocity);
        this.Acceleration = createVector(0,0);
    }

    // Used to draw the particle image at its location to
show the user where the particle is
    display(){
        image(this.Image, this.Location.x, this.Location.y
);
    }

    // Used to calculate a force to a given attractor
    attract(attractor){
        let force = p5.Vector.sub(attractor.Location, this
.Location);
        let distance = force.mag();
        distance = constrain(distance, 60.0, 600.0);
        force.normalize();
        let strength = (this.Mass*attractor.Mass)/pow(dist
ance, 2.0);
        force.mult(strength);
        return force;
    }

    // This function applies the force to the particle usi
ng Newtons F = ma --> a = F / m
    applyForce(force){
        let accel = p5.Vector.div(force, this.Mass);
        this.Acceleration = this.Acceleration.add(accel);
    }

    // This function is used to create an image for the pa
rticle
    createParticleImage(){
        let side = 300;
        let center = 150;

        this.Image = createImage(side, side);

        let num = pow(10, 1.8);

        let h = frameCount;
        let s = this.Color;
        let v = this.Size;
        let c = v * s;
        h = (h % 360) / 60;
        let x = c * (1 - abs(h % 2 - 1));
        let co;
        if (h >= 0 && h < 1) {
```

```
        co = createVector(1, 1, 1).mult(v - c).add(createV
ector(c, x, 0));
        } else if (h >= 1 && h < 2) {
        co = createVector(1, 1, 1).mult(v - c).add(createV
ector(x, c, 0));
        } else if (h >= 2 && h < 3) {
        co = createVector(1, 1, 1).mult(v - c).add(createV
ector(0, c, x));
        } else if (h >= 3 && h < 4) {
        co = createVector(1, 1, 1).mult(v - c).add(createV
ector(0, x, c));
        } else if (h >= 4 && h < 5) {
        co = createVector(1, 1, 1).mult(v - c).add(createV
ector(x, 0, c));
        } else if (h >= 5 && h < 6) {
        co = createVector(1, 1, 1).mult(v - c).add(createV
ector(c, 0, x));
        }

        let Cr = 255 * co.x;
        let Cg = 255 * co.y;
        let Cb = 255 * co.z;

        this.Image.loadPixels();
        for (let i = 0; i < side; i++) {
            for (let j = 0; j < side; j++) {
                let d = (sq(center - j) + sq(center - i))/
num;
```

```
                let col = color(Cr/d, Cg/d, Cb/d);
                this.Image.set(j, i, col);
            }
        }
        this.Image.updatePixels();
        return this.Image;
    }

    // Setters
    set Size(data){
        this.size = data;
    }

    set Color(data){
        this.color = data;
    }

    set Mass(data){
        this.mass = data;
    }

    set Velocity(data){
        this.velocity = data;
    }

    set Location(data){
        this.location = data;
    }

    set Acceleration(data){
        this.acceleration = data;
    }

    set Friction(data){
        this.friction = data;
    }

    set Image(data){
        this.img = data;
    }

    // Getters
    get Color(){
        return this.color;
    }

    get Size(){
        return this.size;
    }

    get Image(){
        return this.img;
    }

    get Location(){
        return this.location;
    }

    get Velocity(){
        return this.velocity;
    }

    get Acceleration(){
        return this.acceleration;
    }

    get Friction(){
        return this.friction;
    }

    get Mass(){
        return this.mass;
    }
}
```

The first function I will talk about is the constructor of the Particle class. The constructor is used to initialise attributes of the class prior to anything else happening within that object. The constructor below is used to define the properties of the particle object upon creation.

As you can see there are a number of parameters being passed to the constructor. The x and y arguments are used to define the location of the particle. the vx and vy arguments are used to define the initial acceleration of the particle. The velocityVector argument is used to define the inital direction and velocity of the particle (this argument is a vector). Finally the sliderValues argument is used to pass a list of slider values that are then used to scale different properties of the particle, so the sliderValues argument is used in the defining of the Color, Size, Friction and Velocity properties of the particle.

Default values were set for some variables, specifically the values which are bound to be different for each particle. I have set default values for all properties containing slider inputs. I have done this just incase the slider for whatever reason may break causing an ‘undefined’, falsey value into the parameters. So if any properties are falsey, they are set to whatever their default value is. Some default values are still zero, which is falsey, but by doing this I am able to catch other types of falsey values like null, undefined and so on and set them to a falsey value that is allowed.

```
constructor(x, y, vx, vy, velocityVector, sliderValues){
    this.Color = sliderValues[0] || 0.8;
    this.Size = sliderValues[1] || 0.8;
    this.Mass = 1;
    this.Friction = 0.1 * sliderValues[4] || 0;
    this.Location = createVector(x, y) || createVector(window.innerWidth/2, window.innerHeight/2);
    this.Velocity = velocityVector.mult(sliderValues[3]) || createVector(0,0);
    this.Acceleration = createVector(vx, vy) || createVector(0,0);
    if(this.Friction == 0.1){this.Friction=1;}
    this.createParticleImage();
}
```

The next function within the particle class is called update. This function is called once every frame. The function is responsible for calculating a velocity from an already known acceleration and then updating that particles location based upon the velocity vector. In short, this function allows the particle to move across the scene.

So to do this, the acceleration vector is added onto the velocity property of the particle and this increases or decreases its velocity depending on the situation, then the velocity is multiplied by a friction value (0 <= friction <= 1) if there is any (this would slow the particles velocity down). Now that the final velocity is calculated, the velocity vector is added onto the location vector, moving the particle along the screen. Finally, the acceleration attribute is set to zero ready for the next update() call.

```
// Used to update the particles velocity and acceleration
and take in account friction (if there is any)
update(){
    this.Velocity = this.Velocity.add(this.Acceleration);
    this.Velocity = this.Velocity.mult(1-this.Friction);
    this.Location = this.Location.add(this.Velocity);
    this.Acceleration = createVector(0,0);
}
```

The next function is also called once every frame and its purpose is to create the particles image as the scene is cleared and re-drawn. This function only calls another function which is passed the Image and Location properties as parameters. It seems pointless having a function just to call another function, but this display function is called once every frame and can be used to call other things if need be.

```
// Used to draw the particle image at its location to show
 the user where the particle is
display(){
    image(this.Image, this.Location.x, this.Location.y);
}
```

The function below is used to calculate a force vector between the particle and a given attractor by taking into account the attractor and particle locations. The distance variable is calculated from the magnitude of the force vector and is then constrained (if the distance value is too low then it will be extremely fast near an attractor, if the distance is too high it will be extremely slow when far away from an attractor). The force vector is then normalised which means that the vector is converted into its unit vector, this allows the script to calculate a custom strength for the particle (the strength is actually based on Newtons law of gravitation). The parameter taken is an attractor obtained from the attractors list in the Input class

```
// Used to calculate a force to a given attractor
attract(attractor){
    let force = p5.Vector.sub(attractor.Location, this.Loc
ation);
    let distance = force.mag();
    distance = constrain(distance, 60.0, 600.0);
    force.normalize();
    let strength = (this.Mass*attractor.Mass)/pow(distance
, 2.0);
    force.mult(strength);
    return force;
}
```

The function below is used to apply a force to the particle (using F=ma, derived from Newtons second law). The force is applied by simply setting a value to the Accelerator attribute. In this case, the acceleration applied to the particle is equal to the force divided by the mass of the particle. The force parameter is a vector which is returned from the attract function within the Particle class.

```
// This function applies the force to the particle using N
ewtons F = ma --> a = F / m
applyForce(force){
    let accel = p5.Vector.div(force, this.Mass);
    this.Acceleration = this.Acceleration.add(accel);
}
```

The function below is used to calculate the colour of the particle. It takes into account the total amount of frames that has passed since launching the game and this determines the colour of the particle at creation, the function also sets a size and colour scale based on the value of the slider at creation time.

```
// This function is used to create an image for the particle
createParticleImage(){
    let side = 300;
    let center = 150;

    this.Image = createImage(side, side);

    let num = pow(10, 1.8);

    let h = frameCount;
    let s = this.Color;
    let v = this.Size;
    let c = v * s;
    h = (h % 360) / 60;
    let x = c * (1 - abs(h % 2 - 1));
    let co;
    if (h >= 0 && h < 1) {
    co = createVector(1, 1, 1).mult(v - c).add(createVector(c, x, 0));
    } else if (h >= 1 && h < 2) {
    co = createVector(1, 1, 1).mult(v - c).add(createVector(x, c, 0));
    } else if (h >= 2 && h < 3) {
```

```
    co = createVector(1, 1, 1).mult(v - c).add(createVecto
r(0, c, x));
    } else if (h >= 3 && h < 4) {
    co = createVector(1, 1, 1).mult(v - c).add(createVecto
r(0, x, c));
    } else if (h >= 4 && h < 5) {
    co = createVector(1, 1, 1).mult(v - c).add(createVecto
r(x, 0, c));
    } else if (h >= 5 && h < 6) {
    co = createVector(1, 1, 1).mult(v - c).add(createVecto
r(c, 0, x));
    }

    let Cr = 255 * co.x;
    let Cg = 255 * co.y;
    let Cb = 255 * co.z;

    this.Image.loadPixels();
    for (let i = 0; i < side; i++) {
        for (let j = 0; j < side; j++) {
            let d = (sq(center - j) + sq(center - i))/num;
            let col = color(Cr/d, Cg/d, Cb/d);
            this.Image.set(j, i, col);
        }
    }
    this.Image.updatePixels();
    return this.Image;
}
```

The code snipped below is all of the getters and setters that are used in the Particle class. I have used getters and setters to ensure no accidental and global changes directly to the property itself. Therefore they can only be accessed from a getter or a setter.

```
    // Setters
    set Size(data){
        this.size = data;
    }

    set Color(data){
        this.color = data;
    }

    set Mass(data){
        this.mass = data;
    }

    set Velocity(data){
        this.velocity = data;
    }

    set Location(data){
        this.location = data;
    }

    set Acceleration(data){
        this.acceleration = data;
    }

    set Friction(data){
        this.friction = data;
    }

    set Image(data){
        this.img = data;
    }

    // Getters
    get Color(){
        return this.color;
    }

    get Size(){
        return this.size;
    }

    get Image(){
        return this.img;
    }

    get Location(){
        return this.location;
    }
    get Velocity(){
        return this.velocity;
    }

    get Acceleration(){
        return this.acceleration;
    }

    get Friction(){
        return this.friction;
    }

    get Mass(){
        return this.mass;
    }
```

The Attractor Class

```
// Used to create new attractor objects
class Attractor{
    constructor(x, y, attractorMassMultiplier){
        this.Mass = 10000.0 * attractorMassMultiplier || 1
0000;
        if(disp.ReverseAttractorDirection){ this.Mass = th
is.Mass * -1; }
        this.Location = createVector(x, y) || createVector
(window.innerWidth/2, window.innerHeight/2);
```

```
        this.display();
    }

    // Used to create the attractor image on the scene
    display(){
        fill(0, 0, 0, 0);
        stroke(200);
        ellipse(this.Location.x, this.Location.y, 5, 5);
    }

    // Setters
    set Mass(data){
        this.mass = data;
    }

    set Location(data){
        this.location = data
    }

    // Getters
    get Mass(){
        return this.mass;
    }

    get Location(){
        return this.location;
    }
}
```

The constructor of the attractor class is used to define the attractors properties and call important functions. In this case, the properties being defined are the mass and the location. The constructor takes three aruments which are x, y and attractorMassMultiplier. x and y specify the location of creation (which is at the position of the mouse) and the attractorMassMultiplier value is the value obtained from the attractor strength slider and is used to scale the attractors gravitational force exerted on a particle.

There is an if statement to check whether or not the attractor should be reversed or not (so more like a repeller, where particles move away from the object). This is done by checking the value of a Boolean variable determined by a button toggle.

Finally, the display function in the class is called to draw it onto the scene.

```
constructor(x, y, attractorMassMultiplier){
    this.Mass = 10000.0 * attractorMassMultiplier || 10000
;
    if(disp.ReverseAttractorDirection){ this.Mass = this.M
ass * -1; }
    this.Location = createVector(x, y) || createVector(win
dow.innerWidth/2, window.innerHeight/2);
    this.display();
}
```

The function below is used to draw the attractor onto the scene so that the user can see its whereabouts. The fill function is used to say that the next shape created is going to be black (the value of black is zero). The stroke function is used to determine the outline of the shape and in this case it is going to be a shade of white (some kind of grey). After the shapes properties have been set the ellipse is drawn at the location of the attractor (obtained from the getters) and has a radius of 5 pixels.

```
// Used to create the attractor image on the scene
display(){
    fill(0, 0, 0, 0);
    stroke(200);
    ellipse(this.Location.x, this.Location.y, 5, 5);
}
```

These are the setters and getters of the Attractor class. They are used as a mask to the main attribute within them, they prevent accidental access to the actual memeber.

```
// Setters
    set Mass(data){
        this.mass = data;
    }

    set Location(data){
        this.location = data
    }

    // Getters
    get Mass(){
        return this.mass;
    }

    get Location(){
        return this.location;
    }
```

The Input Class

```
// Used to handle the inputs
class Input{
    constructor(){
        this.Pos1 = 0;
        this.Pos2 = 0;
        this.InitialVelocity = 0;
        this.ElementClicked = '';
        this.Particles = [];
        this.Attractors = [];
        this.GameHasStarted = false;
    }

    // Used to randomise all particle objects velocities a
nd directions
    randomiseParticles(){
        for(let particle of this.Particles){

            particle.velocity.x = random(-5,5);
            particle.velocity.y = random(-5, 5);
        }
    }

    // Used to handle keyboard presses
    keyboardInput(key) {
        if(this.GameHasStarted){
            if (key === 'p' || key === 'P') {
                this.PushNewParticle = new Particle(mouseX
, mouseY, 0, 0, createVector(random(-5, 5), random(-5, 5))
, disp.SliderValues);
            }
            if (key === 'a' || key === 'A') {
                this.PushNewAttractor = new Attractor(mous
eX, mouseY, disp.SliderValues[2]);
            }
            else if (key === 'r' || key === 'R') {
                this.Particles = [];
                this.Attractors = [];
            }
            else if (key === 'm' || key === 'M') {
                this.randomiseParticles();
            }
        }
    }

    // Used to calculaate velocities for each particle
```

```
    calculateVelocity(){
        let veloc = p5.Vector.sub(this.Pos2, this.Pos1);
        veloc.div(50);
        return veloc;
    }

    // This function is used to update every particle with
 every attractor
    updateObjects(){
        for (let i = 0; i < this.Particles.length; i++) {
            for (let j = 0; j < this.Attractors.length; j+
+) {
                let force = this.Particles[i].attract(this
.Attractors[j]);
                this.Particles[i].applyForce(force);
            }
            this.Particles[i].update();
            this.Particles[i].display();
        }
        for (let i = 0; i < this.Attractors.length; i++) {
                this.Attractors[i].display();
            }
        if(mouseIsPressed && this.ElementClicked === "cnvs
"){
            stroke(255);
            line(this.Pos1.x, this.Pos1.y, mouseX, mouseY)
;
        }
    }

    // Setters
    set PushNewParticle(data){
        this.particles.push(data);
    }

    set PushNewAttractor(data){
        this.attractors.push(data);
    }

    set Pos1(data){
        this.pos1 = data;
    }

    set Pos2(data){
        this.pos2 = data;
    }

    set InitialVelocity(data){
        this.initialVelocity = data;
    }

    set ElementClicked(data){
        this.elementClicked = data;
    }

    set Particles(data){
        this.particles = data;
    }

    set Attractors(data){
        this.attractors = data;
    }

    set GameHasStarted(data){
        this.gameHasStarted = data;
    }

    // Getters
    get Pos1(){
        return this.pos1;
    }

    get Pos2(){
        return this.pos2;
    }

    get InitialVelocity(){
        return this.initialVelocity;
    }

    get ElementClicked(){
        return this.elementClicked;
    }

    get Particles(){
        return this.particles;
    }

    get Attractors(){
        return this.attractors;
    }

    get GameHasStarted(){
        return this.gameHasStarted;
    }
}
```

The constructor class below is of the Input class and takes no parameters. All properties are set by default from code with the use of setters. The Pos1 and Pos2 attributes are used to get the mouse locations at two different points (Pos1 at the mouse press and Pos2 at the mouse release). InitialVelocity is used to pass a velocity vector to the Particle class. The ElementClicked property is used to store the id of an element which the mouse is over on the event of a mouse press. The Particles and Attractors lists are used to store all of the existing particle and attractor objects that are on the scene. Lastly, the GameHasStarted attribute is used to state whether the game has started (used to control whether the homepage covers the game or not).

```
constructor(){
    this.Pos1 = 0;
    this.Pos2 = 0;
    this.InitialVelocity = 0;
    this.ElementClicked = '';
    this.Particles = [];
    this.Attractors = [];
    this.GameHasStarted = false;
}
```

The function in the code snippet below iterates through all of the particle objects on the scene and gives them random velocities and directions. This function is called from another function on a keypress (M).

```
// Used to randomise all particle objects velocities and d
irections
randomiseParticles(){
    for(let particle of this.Particles){
        particle.velocity.x = random(-5,5);
        particle.velocity.y = random(-5, 5);
    }
}
```

The function below handles all keyboard inputs, the argument it is passed is the key that was pressed and comes from the global event function found outside of the class (the parameter comes from outside of the class in a global function called everytime a key is pressed, this is because global events cannot be recognised from within a class without external referencing). When the P key is pressed, a particle object is created at the mouse location with a random velocity and direction. When the A key is pressed an attractor object is created at the location of the mouse pointer and the attractor strength slider value is passed to it. On the press of the R key the scene is reset by simply setting the Particles and Attractors attributes back to an empty list. On the press of the M key, all the particles' velocities and directions are randomised (via the function described in the snippet above). I am testing for these inputs only when the Boolean GameHasStarted is set to true, as I don't want to take input swhen the game has not visually started for the player.

```
// Used to handle keyboard presses
keyboardInput(key) {
    if(this.GameHasStarted){
        if (key === 'p' || key === 'P') {
            this.PushNewParticle = new Particle(mouseX, mo
useY, 0, 0, createVector(random(-5, 5), random(-5, 5)), di
sp.SliderValues);
        }
        if (key === 'a' || key === 'A') {
            this.PushNewAttractor = new Attractor(mouseX,
mouseY, disp.SliderValues[2]);
        }
        else if (key === 'r' || key === 'R') {
            this.Particles = [];
            this.Attractors = [];
        }
        else if (key === 'm' || key === 'M') {
            this.randomiseParticles();
        }
    }
}
```

The snippet of code below is the function used when a particle is created from a mouse drag. This function only happens in the said scenario as only for this scenario do I need to take into account the positions of the mouse at the press of the mouse and the release of the mouse in order to calculate a direction for the particles initial velocity.

```
// Used to calculaate velocities for each particle
calculateVelocity(){
    let veloc = p5.Vector.sub(this.Pos2, this.Pos1);
    veloc.div(50);
    return veloc;
}
```

The function below is called once every frame and is used to draw obejcts on the screen. The first code block is a nested for loop that looks through all particles and then all attractors. For each of these iterations the force of the particle is calculated for that specific attractor is calculated and applied to the particle (via functions mentioned within the Particle class). The particles are then updated and displayed (again by functions within the Particle class). The whole nested loops can be thought of like this, “for each particle, find the force to every attractor”.

After the nested for loops are fnished another for loop is ran and this loop updates the display of each individual attractor.

Lastly, the if statement checks that a mouse button is being pressed and that the elements id which the mouse was initially pressed on is ‘cnvs’. If this condition is true, a white line is drawn from the position which the mouse was pressed up until the final position of where the mouse was released. The whole point of this is to show the user the direction, scale of velocity and the location of the particle they are about to create.

```
// This function is used to update every particle with eve
ry attractor
updateObjects(){
    for (let i = 0; i < this.Particles.length; i++) {
        for (let j = 0; j < this.Attractors.length; j++) {
            let force = this.Particles[i].attract(this.Att
ractors[j]);
            this.Particles[i].applyForce(force);
        }
        this.Particles[i].update();
        this.Particles[i].display();
    }
    for (let i = 0; i < this.Attractors.length; i++) {
            this.Attractors[i].display();
        }
    if(mouseIsPressed && this.ElementClicked === "cnvs"){
        stroke(255);
        line(this.Pos1.x, this.Pos1.y, mouseX, mouseY);
    }
}
```

The code below is all of the setters and getters that I used to access and retrieve attributes within the Input class.

```
// Setters
set PushNewParticle(data){
    this.particles.push(data);
}

set PushNewAttractor(data){
    this.attractors.push(data);
}

set Pos1(data){
    this.pos1 = data;
}

set Pos2(data){
    this.pos2 = data;
}

set InitialVelocity(data){
    this.initialVelocity = data;
}
```

```

set ElementClicked(data){
    this.elementClicked = data;
}

set Particles(data){
    this.particles = data;
}

set Attractors(data){
    this.attractors = data;
}

set GameHasStarted(data){
    this.gameHasStarted = data;
}

// Getters
get Pos1(){
    return this.pos1;
}

get Pos2(){
    return this.pos2;
}

get InitialVelocity(){
    return this.initialVelocity;
}


get ElementClicked(){
    return this.elementClicked;
}


get Particles(){
    return this.particles;
}


get Attractors(){
    return this.attractors;
}


get GameHasStarted(){
    return this.gameHasStarted;
}
```

The Display Class

This class is used to display the slider controls, button, the control text, the help text, the fps, the number of particles and the number of attractors on the scene.

```
class Display{
    constructor(){
        this.ShowControls = true;
        this.ShowHelp = true;
        this.ReverseAttractorDirection = false;
        this.Fr = frameRate();
        this.Buttons = this.setupButtons();
        this.Sliders = this.setupSliders();
    }

    // Used to initialise the buttons and their properties
    setupButtons(){
        let buttonMenu = createButton('Controls');
        buttonMenu.position(window.innerWidth/2- 50,15);
        buttonMenu.mouseClicked(this.toggleControl);
        let buttonHelp = createButton('Help');
        buttonHelp.position(window.innerWidth/2 + 50, 15);
        buttonHelp.mouseClicked(this.toggleHelp);
        let buttonReverseAttraction = createButton('Revers
e Attraction');
        buttonReverseAttraction.position(450, 87);
        buttonReverseAttraction.mouseClicked(this.toggleAt
tractorDirection);
        return [buttonMenu, buttonHelp, buttonReverseAttra
ction];
    }

    // Used to initialise the sliders and their properties
    setupSliders(){
        let sliderColor = createSlider(0, 1, 0.8, 0.1);
        sliderColor.position(290, 7);
        let sliderSize = createSlider(0, 1, 0.8, 0.1);
        sliderSize.position(290, 47);
        let sliderAttractorMass = createSlider(0, 10, 1, 0.5);
        sliderAttractorMass.position(290, 87);
        let sliderVelocity = createSlider(0, 5, 1, 0.1);
        sliderVelocity.position(290, 127);
        let sliderFriction = createSlider(0, 1, 0, 0.01);
        sliderFriction.position(290, 167);
        return [sliderColor, sliderSize, sliderAttractorMass, sliderVelocity, sliderFriction];
    }

    // Used to reset the sliders' values back to their default values prior to user changes
    resetSliders(){
        let sliders = this.Sliders
        sliders[0].value(0.8);
        sliders[1].value(0.8);
        sliders[2].value(1);
        sliders[3].value(1);
        sliders[4].value(0);
    }

    // Used to make all sliders visible to the user
    showSliders(){
        for(let slider of this.Sliders){
            slider.show();
        }
        this.Buttons[2].show();
    }

    // Used to make all slider invisible to the user (and
button)
    hideSliders(){
        for(let slider of this.Sliders){
            slider.hide();
        }
        this.Buttons[2].hide();
    }

    // Updates all necessary information that is displayed
 to the user
    update(){
        this.updateFramerate();
        this.createFooterText();
        if(this.ShowControls){
            this.createControlText();
        }
        if (this.ShowHelp){
            this.createHelpText();
        }
    }

    // Used to create all the text seen in the footer
    createFooterText(){
        textFont('Monospace');
        textSize(24);
        fill("white");
        textAlign(RIGHT, BOTTOM);
        text(str(int(this.fr))+" fps", width, height);
        textAlign(LEFT, BOTTOM);
        text("Attractor : "+str(inp.attractors.length), 0,
 height);
        textAlign(CENTER, BOTTOM);
        text("Particle : "+str(inp.particles.length), widt
h / 2, height);
        textAlign(LEFT, CENTER);
    }

    // Creates text to display help to the user
    createHelpText(){
        text("To create a particle you can either:", 10, 2
40);
        text("  1. Press P (creates a particle with a rand
om direction and velocity at the position of the cursor)."
, 10, 280);
        text("  2. Click and drag to create a particle whi
ch has direction of the drag line", 10, 320)
        text("     and the velocity with respect to the si
ze of the drag distance.", 10, 360);
        text("To create an attractor press the A key.", 10
, 400);
        text("to randomise ALL particle directions and vel
ocities press the M key.", 10, 440);
```

```
        text("To reset the scene press the R key.", 10, 480);
        text("The slider controls at the top left control the properties:", 10, 520);
        text("  1. The size slider adjusts the visual size of any new particle.", 10, 560);
        text("  2. The color slider allows for a variety of different coloured particles.", 10, 600);
        text("  3. The attra. strength slider allows for the effect of attractors with more gravitational force.", 10, 640);
        text("  4. The velocity slider adjusts the velocity multiplier of any new particle.", 10, 680);
        text("  5. The friction slider adds a 'drag' on any new particles.", 10, 720);
        text("  6. The REVERSE ATTRACTION button is used to repel particles from any new attractor.", 10, 760);
        text("To close this help, press the HELP button at the top. To close the slider controls, press the CONTROLS button at the top.", 10, 800);
        text("To go back to the home page of the website, press Escape (ESC)", 10, 840);
    }

    // Creates the text for the control sliders
    createControlText(){
        text("Color          : "+str(this.SliderValues[0]), 10, 20);
        text("Size           : "+str(this.SliderValues[1])
, 10, 60);
        text("Attra. Strength: "+str(this.SliderValues[2])
, 10, 100);
        if(disp.ReverseAttractorDirection){
            fill("green");
            text("ON", 585, 100);
        } else {
            fill("red");
            text("OFF", 585, 100);
        }
        fill("white");
        text("Velocity       : "+str(this.SliderValues[3])
, 10, 140);
        text("Friction       : "+str(this.SliderValues[4])
, 10, 180);
    }

    // Returns framerate value after every 10 frames
    updateFramerate(){
        if (frameCount % 10 == 0) {
            this.Fr = frameRate();
        }
    }

    // Used to switch a Boolean indicating whether or not
to show sliders
    toggleControl(){
        if (disp.ShowControls){
            disp.hideSliders();
            disp.ShowControls = false;
        }else{
            disp.showSliders();
            disp.ShowControls = true;
        }
    }

    // Used to switch a Boolean indicating whether or not
to show help
    toggleHelp(){
        if(disp.ShowHelp){
            disp.ShowHelp = false;
        } else{
            disp.ShowHelp = true;
        }
    }

    // Used to switch a Boolean indicating whether the new
ly created attractors are repellers or not
    toggleAttractorDirection(){
        if(disp.ReverseAttractorDirection){
            disp.ReverseAttractorDirection = false;
        }else{
            disp.ReverseAttractorDirection = true;
        }
    }
```

```
// Setters
set ShowControls(data){
    this.showControls = data;
}

set ShowHelp(data){
    this.showHelp = data;
}

set Fr(data){
    this.fr = data;
}

set Sliders(data){
    this.sliders = data;
}

set Buttons(data){
    this.buttons = data;
}

set ReverseAttractorDirection(data){
    this.reverseAttractorDirection = data;
}

// Getters
```

```
    get ShowControls(){
        return this.showControls;
    }

    get ShowHelp(){
        return this.showHelp;
    }

    get Fr(){
        return this.fr;
    }

    get Sliders(){
        return this.sliders;
    }

    get Buttons(){
        return this.buttons;
    }

    get SliderValues(){
        return [this.Sliders[0].value(), this.Sliders[1].v
alue(), this.Sliders[2].value(), this.Sliders[3].value(),
this.Sliders[4].value()]
    }

    get ReverseAttractorDirection(){
        return this.reverseAttractorDirection;
    }
}
```

The constructor of the Display class (below) does not recieve any parameters, the attributes are set by global functions or from functions within the class. The ShowControls attribute is used to say whether the control sliders and text should be displayed or not. The ShowHelp attribute is used to say whether the help text should be displayed or not. The ReverseAttractorDirection attribute is used to say whether newly created attractors should be repellers or not. The Fr attribute (short for FrameRate) is used to store the framerate at the time of object creation. The Buttons and Sliders attributes are used to store all buttons and sliders elements respectivly.

```
constructor(){
    this.ShowControls = true;
    this.ShowHelp = true;
    this.ReverseAttractorDirection = false;
    this.Fr = frameRate();
    this.Buttons = this.setupButtons();
    this.Sliders = this.setupSliders();
}
```

The function below is called once from the constructor and is used to create 3 buttons on the scene. The first button is a button with the label ‘Controls’ which is position at the top-centre and when pressed it calls a function named toggleControl (located in the same class as shown by the use of ‘this’). The second button created has the label ‘Help’, it is positioned at the top-centre aswell and when pressed it calls a function named ‘toggleHelp’. The last button created has label ‘Reverse Attraction’ and is positioned just to the right of the attractor strength slider (near the top-left of the scene) and when this button is pressed it calls a function named ‘toggleAttractorDirection’. Finally all of the button elements are returned to the caller (which is in the contructor).

```
// Used to initialise the buttons and their properties
setupButtons(){
    let buttonMenu = createButton('Controls');
    buttonMenu.position(window.innerWidth/2- 50,15);
    buttonMenu.mouseClicked(this.toggleControl);
    let buttonHelp = createButton('Help');
    buttonHelp.position(window.innerWidth/2 + 50, 15);
    buttonHelp.mouseClicked(this.toggleHelp);
    let buttonReverseAttraction = createButton('Reverse At
traction');
    buttonReverseAttraction.position(450, 87);
    buttonReverseAttraction.mouseClicked(this.toggleAttrac
torDirection);
    return [buttonMenu, buttonHelp, buttonReverseAttractio
n];
}
```

The code snippet below is of a function used to create slider elements. The five sliders created are to change the colour of the particle, the size of the particle, the mass of the attractor (strength), the velocity multiplier of the particle and finally the friction of the particle. Each slider has a different minimum value, maximum value, default value and step value. For example, the sliderVelocity slider has a min value of 0, a max value of 5, a default value of 1 and a step of 0.1. They are all position in the top-left of the scene in the order they are defined in. The slider elements are returned to its caller (to the constructor).

```
// Used to initialise the sliders and their properties
setupSliders(){
    let sliderColor = createSlider(0, 1, 0.8, 0.1);
    sliderColor.position(290, 7);
    let sliderSize = createSlider(0, 1, 0.8, 0.1);
    sliderSize.position(290, 47);
    let sliderAttractorMass = createSlider(0, 10, 1, 0.5);
    sliderAttractorMass.position(290, 87);
    let sliderVelocity = createSlider(0, 5, 1, 0.1);
    sliderVelocity.position(290, 127);
    let sliderFriction = createSlider(0, 1, 0, 0.01);
    sliderFriction.position(290, 167);
    return [sliderColor, sliderSize, sliderAttractorMass,
sliderVelocity, sliderFriction];
}
```

The function below is used to reset the sliders values back to their defaults. This function is called when the escape button is pressed, which is to go back to the homepage.

```
// Used to reset the sliders' values back to their default
 values prior to user changes
resetSliders(){
    let sliders = this.Sliders
    sliders[0].value(0.8);
    sliders[1].value(0.8);
    sliders[2].value(1);
    sliders[3].value(1);
    sliders[4].value(0);
}
```

The two blocks of code below are used to show sliders and hide sliders (and the button with the sliders) respectfully. These are the functions that are ran when the ShowSliders attribute is true for showSliders() and false for hideSliders(). They simply loop through the Sliders list and apply the .show() or .hide() function to them depending on the executed function.

```
// Used to make all sliders visible to the user
showSliders(){
    for(let slider of this.Sliders){
        slider.show();
    }
    this.Buttons[2].show();
}

// Used to make all slider invisible to the user (and butt
on)
hideSliders(){
    for(let slider of this.Sliders){
        slider.hide();
    }
    this.Buttons[2].hide();
}
```

The function below is called once every frame and is used to update the Fr (framerate value), show the text at the bottom of the page (contains attractors, particles and fps counter), determine whether or not to show the controls text and also determines whether or not to show the help text onto the scene. All of which are done by functions local to the class.

```
// Updates all necessary information that is displayed to
the user
update(){
    this.updateFramerate();
    this.createFooterText();
    if(this.ShowControls){
        this.createControlText();
    }
    if (this.ShowHelp){
        this.createHelpText();
    }
}
```

The two functions below are used to create the text displayed at the footer of the scene and the help information respectivly. The font style, size and colour are all set in the createFooterText function (as this is the first function called).

```
// Used to create all the text seen in the footer
createFooterText(){
    textFont('Monospace');
    textSize(24);
    fill("white");
    textAlign(RIGHT, BOTTOM);
    text(str(int(this.fr))+" fps", width, height);
    textAlign(LEFT, BOTTOM);
    text("Attractor : "+str(inp.attractors.length), 0, height);
    textAlign(CENTER, BOTTOM);
    text("Particle : "+str(inp.particles.length), width / 2, height);
    textAlign(LEFT, CENTER);
}

// Creates text to display help to the user
createHelpText(){
    text("To create a particle you can either:", 10, 240);
    text("  1. Press P (creates a particle with a random direction and velocity at the position of the cursor).", 10, 280);
    text("  2. Click and drag to create a particle which has direction of the drag line", 10, 320)
    text("     and the velocity with respect to the size of the drag distance.", 10, 360);
    text("To create an attractor press the A key.", 10, 400);
    text("to randomise ALL particle directions and velocities press the M key.", 10, 440);
    text("To reset the scene press the R key.", 10, 480);
    text("The slider controls at the top left control the properties:", 10, 520);
    text("  1. The size slider adjusts the visual size of any new particle.", 10, 560);
    text("  2. The color slider allows for a variety of different coloured particles.", 10, 600);
    text("  3. The attra. strength slider allows for the effect of attractors with more gravitational force.", 10, 640);
    text("  4. The velocity slider adjusts the velocity multiplier of any new particle.", 10, 680);
    text("  5. The friction slider adds a 'drag' on any new particles.", 10, 720);
    text("  6. The REVERSE ATTRACTION button is used to repel particles from any new attractor.", 10, 760);
    text("To close this help, press the HELP button at the top. To close the slider controls, press the CONTROLS button at the top.", 10, 800);
    text("To go back to the home page of the website, pres
s Escape (ESC)", 10, 840);
}
```

The function below contains the code used to display the control / slider text and their current values. The values are obtained from the getter from the same class as the fucntion is in. It is also used to display green ON text or red OFF text next to the reverse attraction button depending on its state.

```
// Creates the text for the control sliders
createControlText(){
    text("Color           : "+str(this.SliderValues[0]), 10
, 20);
    text("Size            : "+str(this.SliderValues[1]), 10
, 60);
    text("Attra. Strength: "+str(this.SliderValues[2]), 10
, 100);
    if(disp.ReverseAttractorDirection){
        fill("green");
        text("ON", 585, 100);
    } else {
        fill("red");
        text("OFF", 585, 100);
    }
    fill("white");
    text("Velocity        : "+str(this.SliderValues[3]), 10
, 140);
```

```
    text("Friction       : "+str(this.SliderValues[4]), 10
, 180);
}
```

The small function below is called every frame but only every makes a change to the Fr (framerate) attribute when the total amount of frames is a multiple of 10, then only will the shown framerate change.

```
// Returns framerate value after every 10 frames
updateFramerate(){
    if (frameCount % 10 == 0) {
        this.Fr = frameRate();
    }
}
```

The three functions below are used to toggle between the two available Boolean states. If a Boolean value is true, then when the function is called it will be set to false and vice-versa. These are the functions called on the press of the three available buttons.

```
// Used to switch a Boolean indicating whether or not to s
how sliders
toggleControl(){
    if (disp.ShowControls){
        disp.hideSliders();
        disp.ShowControls = false;
    }else{
        disp.showSliders();
        disp.ShowControls = true;
    }
}

// Used to switch a Boolean indicating whether or not to s
how help
toggleHelp(){
    if(disp.ShowHelp){
        disp.ShowHelp = false;
    } else{
        disp.ShowHelp = true;
    }
}

// Used to switch a Boolean indicating whether the newly c
reated attractors are repellers or not
toggleAttractorDirection(){
    if(disp.ReverseAttractorDirection){
        disp.ReverseAttractorDirection = false;
    }else{
        disp.ReverseAttractorDirection = true;
    }
}
```

The code snippet below is of the available getters and setters in the Display class.

```
// Setters
set ShowControls(data){
    this.showControls = data;
}

set ShowHelp(data){
    this.showHelp = data;
}

set Fr(data){
    this.fr = data;
}

set Sliders(data){
    this.sliders = data;
}

set Buttons(data){
    this.buttons = data;
}

set ReverseAttractorDirection(data){
    this.reverseAttractorDirection = data;
}

// Getters

```

```
get ShowControls(){
    return this.showControls;
}

get ShowHelp(){
    return this.showHelp;
}

get Fr(){
    return this.fr;
}

get Sliders(){
    return this.sliders;
}

get Buttons(){
    return this.buttons;
}

get SliderValues(){
    return [this.Sliders[0].value(), this.Sliders[1].value
(), this.Sliders[2].value(), this.Sliders[3].value(), this
.Sliders[4].value()]
}

get ReverseAttractorDirection(){
    return this.reverseAttractorDirection;
}
```

The Rest of the Script

This part of the script is not contained in classes as the code here is either global events or required by global events (global events cannot be handled from within a class as the object for that class may not be created).

The two variables, inp and dip, are used to store the instances of the classes Input and Display. These are global variables as they are accessed throughout the whole script.

The function named setup() is a global function and is called only once when the page has loaded. This setup assigns inp and disp an instance of Input and Disp. The z-index property of the ‘header’ `div` is set to 1 so that everything on the canvas (scene) appears behind the homepage at startup (z-index is like the z axis into the screen where 1 is infront of anything with a index lower than it). Then the canvas is created with a width and height equal to that of the browser and is given the id of ‘cnvs’. The margins of the body tags are set to 0 and the padding of the canvas is set to 0 to eliminate any borders that may appear. Setting the overflow property to hidden removes any scroll bars that may occur.

Now an event listner is added and is ran anytime a mouse button is pressed. The elements id which the mouse cursor is hovering over is sent to the Input class to be stored and used later. Another event listner is added and this one runs anytime a key is pressed on the keyboard. It checks to see if the key pressed was key 27 (which is equivelent to the ESCAPE button) and if it is, the Particles and Attractors list from the Input class are set to empty, the function resetSliders is called from the Display class and finally the transitionToHomepage is called to bring back the homepage.

The blendMode function is called (part of the p5js library) and is used to blend the pixels in the display to the ADD mode. The ADD mode means to use additive blending with a white clip. The imageMode function is also part of p5js and is used to CENTER the image within its canvas in this case.

The last two functions called in setup is frameRate(60) and background(0) and are used to cap the frameRate at 60 and set the background colour to black initally.

```
// Declaring variables that are accessed throughout the sc
ript
let inp;
let disp;

// The first function called, used to get all the importat
ing setting and properties set before the user interacts
function setup() {
    inp =  new Input();
    disp = new Display();
    document.getElementById("header").style.zIndex = "1";
    let scene = createCanvas(window.innerWidth, window.inn
erHeight);
    scene.id('cnvs');
    document.body.style.margin = 0;
    document.getElementById('cnvs').style.padding = 0;
    document.body.style.overflow = 'hidden';
    document.addEventListener('mousedown', function(e) {
            inp.ElementClicked = e.target.id;
        }, false);
    document.onkeydown = function(evt) {
        evt = evt || window.event;
        if (evt.keyCode == 27) {
            transitionToHomepage();
            disp.resetSliders();
            inp.Particles = [];
            inp.Attractors = [];
        }
    };
    blendMode(ADD);
    imageMode(CENTER);
    frameRate(60);
    background(0);
}
```

The draw function is called once per frame and is a standard function of the p5js library. It is important to clear the canvas before drawing anything back onto the canvas otherwise old shapes etc. will still remain on the screen (like all previous particle locations). As the canvas is cleared, background colour, and everything else on the screen will have been removed. Therefore, bakground colour is set back to black, and the display and input classes are updated.

```
// Function called once per frame
function draw() {
  clear();
  background(0);
  disp.update();
  inp.updateObjects();
}
```

The two functions below are global functions and are called whenever the mouse is pressed or released respectfully. They are used to set the Pos1 (a setter in the Input class) and Pos2 (a setter in the Input class) and then create a particle based on those calculations. Do note the particle is only created when the initial mouse press is on the element with the id ‘cnvs’ which is the canvas.

```
// A global function called when any of the mouse buttons
are pressed
function mousePressed(){
    inp.Pos1 = createVector(mouseX, mouseY);
}

// A global function called when any of the mouse buttons
that are held, are suddenly released
```

```
function mouseReleased(){
    inp.Pos2 = createVector(mouseX, mouseY);
    if(inp.ElementClicked == 'cnvs'){ inp.PushNewParticle
= new Particle(inp.Pos1.x, inp.Pos1.y, 0, 0, inp.calculate
Velocity(), disp.SliderValues); }
}
```

The code below is of two global functions. The windowResized() function is called in the event of the browser being resized. This fucntion sets the canvas size equal to the browser size, keeping it fullscreen. The two central control and help buttns are also repositioned to the centre of the screen after the resize.

The keyTyped function is called in the event of a keypress and the key that is pressed is send to the keyboardInput function within the inp object (Input class) to be handled over there.

```
// A global function called everytime the browser window i
s resized
function windowResized() {
    resizeCanvas(window.innerWidth, window.innerHeight);
    disp.Buttons[0].position(window.innerWidth/2- 50,15);
    disp.Buttons[1].position(window.innerWidth/2 + 50, 15)
;
}

// A global function called everytime a key is pressed
```

```
function keyTyped() {
    inp.keyboardInput(key);
}
```

The two functions below are responsible for moving the hompage `div` out of the way so that the game can be seen or putting the homepage back so that it cant be seen. This is done by a transition that takes 2 seconds and moves the left margin to -100% its current position, completely making it move off of the screen. The GameHasStarted attribute in the inp object is set to true so that to allow inputs to be handled. The opposite happens for the transitionToHomepage() function which moves the homepage `div` back into the screen.

```
// Function used to transition from the home page to the scene
function transitionToGame(){
    document.getElementById("header").style.transition = "all 2s";
    document.getElementById("header").style.marginLeft = "-100%";
    inp.GameHasStarted = true;
}

// Function used to transition from the scene back to the home page
function transitionToHomepage(){
    document.getElementById("header").style.transition = "
all 2s";
    document.getElementById("header").style.marginLeft = "
0";
    inp.GameHasStarted = false;
}
```