Boomshine-E

Overview:
Time to refactor and add a few more features. This time we'll fix the circle color, add sound, detect key presses, create a module, and add a particle system to our *Boomshine* clone.

Part I - A little Refactoring

Make a copy of the **Boomshine-D folder** and name it **Boomshine-E**

1) Why is the circle we click turning solid red? Delete the following line of code in `app.main.checkCircleClicked()`:

```
c.fillStyle = "red";
```

2) Remember the crappy code that was left in `doMousedown()` that we hoped you would fix:

```
// ugh - the 'this' issue - should have planned ahead better
var main = app.main;
```

Did you fix it yet? If not, let's go ahead and do so:

A) In `app.main.init()`, change this line:

```
this.canvas.onmousedown = this.doMousedown;
```

to this:

```
this.canvas.onmousedown = this.doMousedown.bind(this);
```

In the first line of code, inside of `doMouseDown()` when it is called, `this` will be equal to the `<canvas>` element.

`this.doMousedown.bind(this)` makes `this` whatever value you pass in, which in this case is app.main.

In the second line of code, inside of `doMouseDown()` when it is called, `this` will be equal to `app.main` (which is thanks to `Function.bind()`)

** We discussed this back in Boomshine-A, that the value of this is determined by how the function was ***called***. **

B) In `doMousedown()` delete this line:

```
var main = app.main;
```

C) Now you can replace "`main`" and "`app.main`" with "`this`" in the rest of the `doMousedown()` method.

D) Reload and test the game by playing a few levels.

3) Our first drawing method in `app.main` was the `fillText()` method. You might have noticed that all of our other drawing methods takes a `ctx` object as the first argument. We did that because then the same method could be used to draw to any graphics context, even an offscreen one, which makes it more reusable. (it also means typing `this` a lot less in the method)

```
fillText: function(ctx,string, x, y, css, color) {
    ctx.save();
    // https://developer.mozilla.org/en-US/docs/Web/CSS/font
    ctx.font = css;
    ctx.fillStyle = color;
    ctx.fillText(string, x, y);
    ctx.restore();
},
```

A) Let's modify `fillText()` to use the same convention. Make it look like this:

B) Now you will need to add `this.ctx` as the first argument of all of the `this.fillText(…)` calls in the code. There should be 8 such places.

4) Some of these randomly colored circles actually don't look that great. Let's instead create an array of colors we know look OK.

A) Add the code below as a property of `app.main` (you can copy/paste from the file in mycourses)

```
// original 8 fluorescent crayons: https://en.wikipedia.org/wiki/List_of_Crayola_crayon_colors#Fluorescent_crayons
//  "Ultra Red", "Ultra Orange", "Ultra Yellow","Chartreuse","Ultra Green","Ultra Blue","Ultra Pink","Hot Magenta"
colors: ["#FD5B78","#FF6037","#FF9966","#FFFF66","#66FF66","#50BFE6","#FF6EFF","#EE34D2"],
```

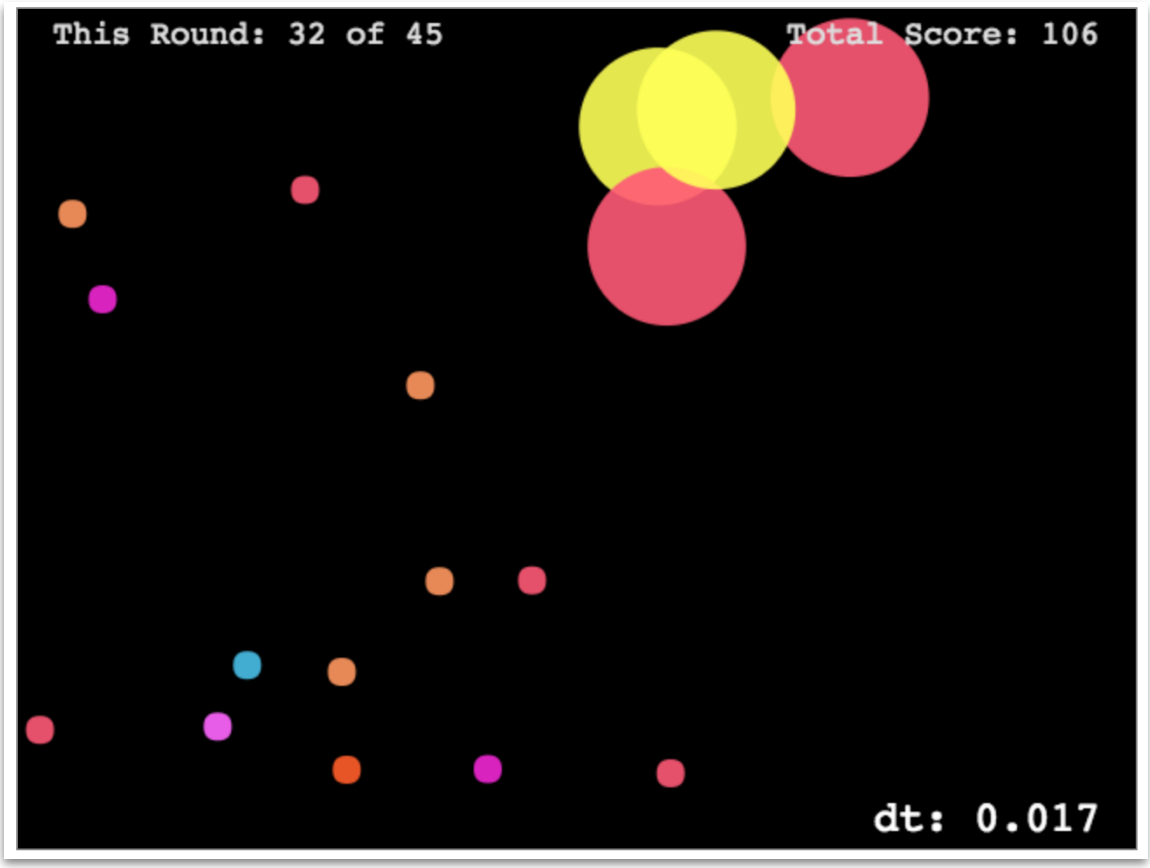
B) In `app.main.makeCircles()`, change this line:

```
c.fillStyle = getRandomColor();
```

to this:

```
c.fillStyle = this.colors[i % this.colors.length];
```

C) Reload the page - fluorescent colored circles!



5) You may occasionally get some “edge bouncing” with a circle caught along one side or another, bouncing back and forth due to Number rounding issues.An easy and cheap fix for that is to give the circles and extra move after they’ve hit a wall and we’ve switched their direction. It also gives a noticeable “pop” or bounce to their motion - it’s a nice effect.

Change these lines in `app.main.update()`:

```
// did circles leave screen?
if(this.circleHitLeftRight(c)) c.xSpeed *= -1;
if(this.circleHitTopBottom(c)) c.ySpeed *= -1;
```

to this:

```
// did circles leave screen?
if(this.circleHitLeftRight(c)){
	c.xSpeed *= -1;
	c.move(dt); // an extra move
}

if(this.circleHitTopBottom(c)){
	c.ySpeed *= -1;
	c.move(dt); // an extra move
}
```

6) We've got some external code (the **loader.js** script) directly setting properties of `app.main` - in this case `app.main.paused`. Let's instead have that code call methods of `app.main` that toggles the paused property for us.

A) Create a `pauseGame()` method in `app.main` that looks like this:

```
pauseGame: function(){

      this.paused = true;

      // stop the animation loop
      cancelAnimationFrame(this.animationID);

      // call update() once so that our paused screen gets drawn
      this.update();
},
```

B) Create an `resumeGame()` method in `app.main` that looks like this:

```
resumeGame: function(){

      // stop the animation loop, just in case it's running
      cancelAnimationFrame(this.animationID);

      this.paused = false;

      // restart the loop
      this.update();
}
```

C) Now the blur and focus methods in **loader.js** can get cleaned up - they should appear as follows:

```
window.onblur = function(){
      console.log("blur at " + Date());
      app.main.pauseGame();
};

window.onfocus = function(){
      console.log("focus at " + Date());
      app.main.resumeGame();
};
```

Part II - Background Sound

Boomshine doesn't have heavy sound requirements, so we're going to do sound the super easy way here just using a couple of `<audio>` tags.

Important! This technique won't work if we have 500 circles and a lot of circles exploding at the same time, and the browser might crash. (Later we'll see how to use the **Sound.js** library to get more sounds playing simultaneously)

1) Download the **media** folder from mycourses and put it in your project folder.

2) Add the following HTML to the `<body>` tag of **boomshine.html**
(this is in mycourses)

```
<section id="audioControls">
      <audio id="bgAudio" src="media/background.mp3" controls loop></audio>
</section>
```

3) The user doesn't need to see these controls - so hide them. Add this CSS to the `<style>` tag of **boomshine.html**:

```
#audioControls{ display:none;}
```

4) Add the following properties to `app.main`:

```
bgAudio: undefined,
currentEffect : 0,
currentDirection : 1,
effectSounds
:["1.mp3","2.mp3","3.mp3","4.mp3","5.mp3","6.mp3","7.mp3","8.mp3"],
```

5) Add the following to `app.main.init()`:

```
this.bgAudio = document.querySelector("#bgAudio");
this.bgAudio.volume=0.25;
```

6) Add a helper function to stop the background audio:

```
stopBGAudio: function(){
     this.bgAudio.pause();
     this.bgAudio.currentTime = 0;
}
```

7) Add this to the code in the `app.main.resumeGame()`:

```
this.bgAudio.play();
```

add this to the top of `app.main.doMouseDown()`:

```
this.bgAudio.play();
```

8) Reload the page. Once you click in the window, the music should start.

9) To turn off the music when the browser is paused, add this to `app.main.pauseGame()`:

```
this.stopBGAudio();
```

10) To turn off the music between levels, call `this.stopBGAudio();`

inside of the `if (isOver){}` block in `app.main.checkForCollisions()`

11) Reload and test it. The background music should now stop between levels, and whenever the app is paused.

Part III - Effect Sound

For the effect sounds, we'll create those dynamically since we could have many. Each time we play one, we'll increase the pitch, until we run out of sounds.

1) Add this method to `app.main`:

```
function playEffect(){
  var effectSound = document.createElement('audio');
  effectSound.volume = 0.3;
  effectSound.src = "media/" + effectSounds[currentEffect];
  effectSound.play();
  currentEffect += currentDirection;
  if (currentEffect == effectSounds.length || currentEffect == -1){
    currentDirection *= -1;
    currentEffect += currentDirection;
  }
}
```

2) Call `this.playEffect()`:

- when a circle is clicked
- when circles intersect

3) Reload the page and test it. You should hear the effect sound play at the appropriate moments.

Part IV - Keyboard Control - the keyup event

We could definitely get by without keyboard controls in Boomshine, but we're going to cover it anyway because many web browser games & apps rely on them.

1) Download the **keys.js** script from mycourses and put it in your **js** folder.

It looks like this:

```
// The myKeys object will be in the global scope - it makes this script
// really easy to reuse between projects

"use strict";

var myKeys = {};

myKeys.KEYBOARD = Object.freeze({
    "KEY_LEFT": 37,
    "KEY_UP": 38,
    "KEY_RIGHT": 39,
    "KEY_DOWN": 40,
    "KEY_SPACE": 32,
    "KEY_SHIFT": 16
});

// myKeys.keydown array to keep track of which keys are down
// this is called a "key daemon"
// main.js will "poll" this array every frame
// this works because JS has "sparse arrays" - not every language does
myKeys.keydown = [];


// event listeners
window.addEventListener("keydown",function(e){
    console.log("keydown=" + e.keyCode);
    myKeys.keydown[e.keyCode] = true;
});

window.addEventListener("keyup",function(e){
    console.log("keyup=" + e.keyCode);
    myKeys.keydown[e.keyCode] = false;

    // pausing and resuming
    var char = String.fromCharCode(e.keyCode);
    if (char == "p" || char == "P"){
        if (app.main.paused){
            app.main.resumeGame();
        } else {
            app.main.pauseGame();
        }
    }
});
```

2) Write a `<script>` tag in **boomshine.html** that links to it.

3) Reload the page and press the ‘p’ key - it should toggle pausing on and off.

4) Look over the code - note the “pause and resume” section of the “keyup” part of the code - hopefully what’s going on there is self explanatory.

Let’s move on to the `mykeys.keydown` array.

Part V - Keyboard Control - detecting more than one key down at a time

1) `mykeys.keydown` is an array that will keep track of which keys are being held down at any given time.

example: If the user presses the **left key** (keycode 37) and the **up key** (keycode 38) down at the same time, then the event listeners will fire and set 2 values in `mykeys.keydown`

```
mykeys.keydown[37] = true
mykeys.keydown[38] = true
```

if the user lifts their finger off of the **up key**, then the event listeners will fire and set 1 value in `mykeys.keydown`

```
mykeys.keydown[38] = false
```

and `mykeys.keydown[37]` is still true

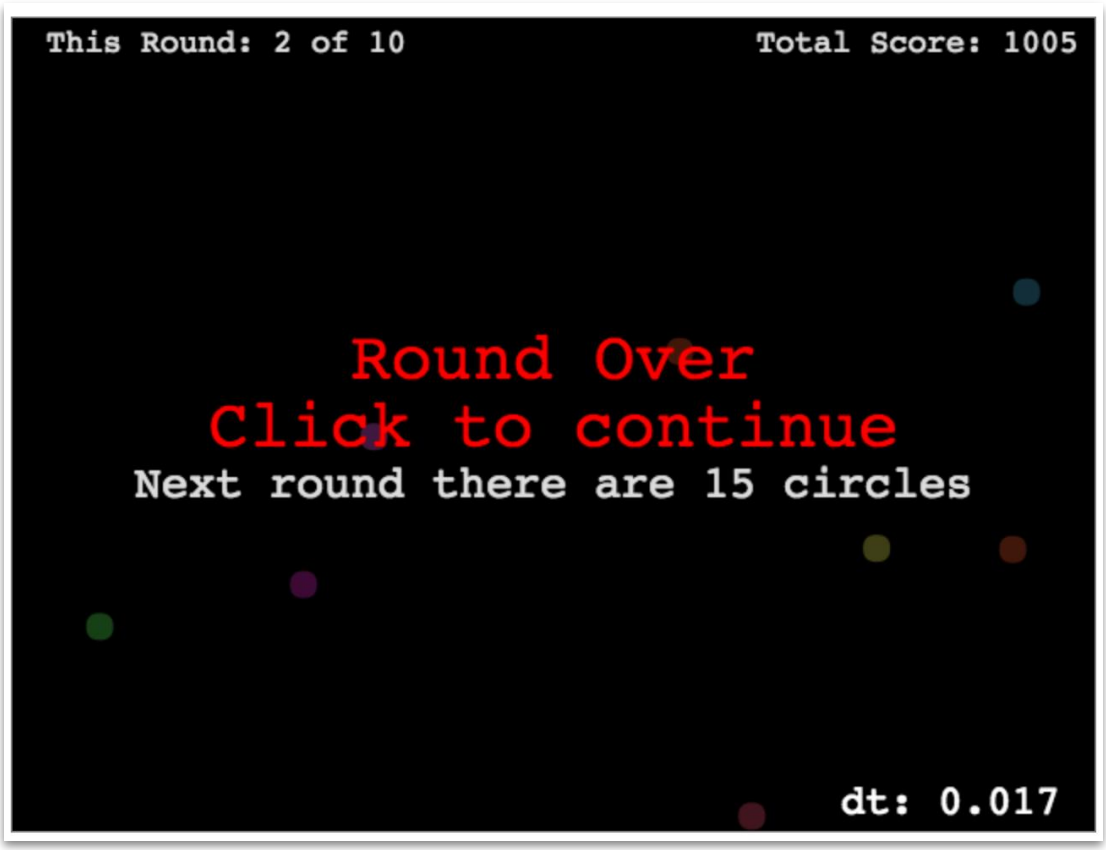
Such an array for tracking keys is called a *key daemon*

2) Now our code in `app.main` is going to *poll* the `mykeys.keydown` array every `update()` call (ie. every frame of the animation) and do a cheat for the player (increase their score) if the right key combination is held down at the right time.

```
// 6) CHECK FOR CHEATS
// if we are on the start screen or a round over screen
if(this.gameState == this.GAME_STATE.BEGIN || this.gameState == this.GAME_STATE.ROUND_OVER){
    // if the shift key and up arrow are both down (true)
    if (myKeys.keydown[myKeys.KEYBOARD.KEY_UP] && myKeys.keydown[myKeys.KEYBOARD.KEY_SHIFT] ){
        this.totalScore ++;
        this.playEffect();
    }
}
```

Add the following to `app.main.update()`:

3) Test it - you should be able to increase your score from either the start screen or the between level screen, by holding down the **shift-up arrow** key.



If you are going to do smooth keyboard control in your game, for moving a spaceship or a paddle for example, you will definitely use this array *polling* technique. It's much smoother than trying to animate on jerky `keyup` events.

Part VI - Where's the modules?

What's a *module* by the way? A module is a bunch of JavaScript code that is executed inside of an IIFE, thus allowing it to have private state and functions that are hidden from the outside. To be useful to us, at least part of the module needs to be exposed to other parts of our JavaScript program.

Reminder: An IIFE is an anonymous JavaScript function that gets executed right when it is declared.

Why modules? Splitting up the code between files makes the project easier to build and maintain, and easier to find bugs. It also makes it simpler to farm out work between multiple developers.

You might have noticed that we started this whole Boomshine exercise talking about modules, but you might think we haven't actually yet created any new modules.

utilities.js isn't a module - it contains generic JavaScript functions that we will reuse between projects. All of the se functions are in the global scope.

loader.js isn't a module - it's actually a sandbox that creates the `app` global for us, and will soon be used to connect other modules together.

main.js IS a module, and is a property of our `app` global - app.main

keys.js IS a module, but we created it as the `myKeys` global object, it is not a property of `app.main`. We did this here to make it easily reusable between projects, but if you wanted you could make it a property of `app` like `main` is.

1) Let's create a **sound.js** module to contain the sound code we made earlier!

A) Create a new file named **sound.js** and put in in the **js** folder.

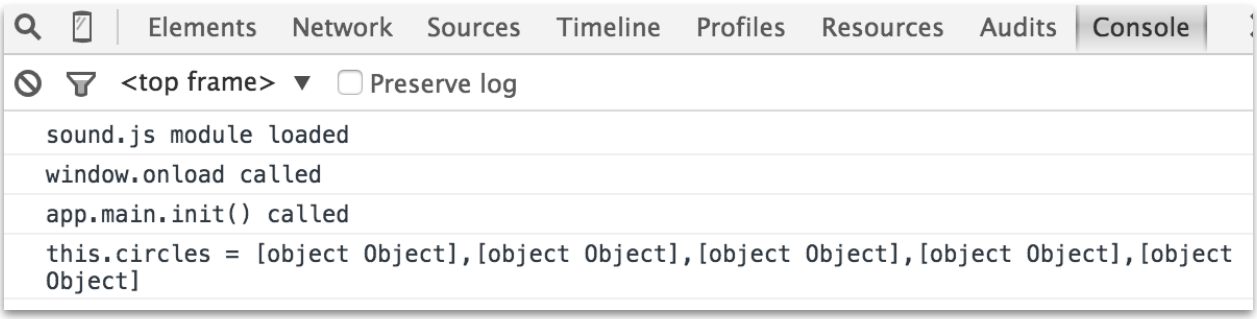
```
// sound.js
"use strict";
// if app exists use the existing copy
// else create a new object literal
var app = app || {};

// define the .sound module and immediately invoke it in an IIFE
app.sound = (function(){
    console.log("sound.js module loaded");
    // TODO
}());
```

B) Create a quick stub of our **sound.js** module code:

C) In **boomshine.html** import the script:

```
<script src='js/sound.js'></script>
```

D) Reload the page. You should see from the log that the **sound.js** module has loaded



E) Here's are starter **sound.js** - go download it from mycourses - the code is similar to the sound from **main.js** - but don't delete any code in **main.js** yet.

```
// sound.js
"use strict";
// if app exists use the existing copy
// else create a new object literal
var app = app || {};

// define the .sound module and immediately invoke it in an IIFE
app.sound = (function(){
  console.log("sound.js module loaded");
  var bgAudio = undefined;
  var currentEffect = 0;
  var currentDirection = 1;
  var effectSounds = ["1.mp3","2.mp3","3.mp3","4.mp3","5.mp3","6.mp3","7.mp3","8.mp3"];

  function init(){
    bgAudio = document.querySelector("#bgAudio");
    bgAudio.volume=0.25;
  }

  function stopBGAudio(){
    bgAudio.pause();
    bgAudio.currentTime = 0;
  }

  function playEffect(){
    var effectSound = document.createElement('audio');
    effectSound.volume = 0.3;
    effectSound.src = "media/" + effectSounds[currentEffect];
    effectSound.play();
    currentEffect += currentDirection;
    if (currentEffect == effectSounds.length || currentEffect == -1){
      currentDirection *= -1;
      currentEffect += currentDirection;
    }
  }

  // export a public interface to this module
  //TODO

}());
```

So you now have a `sound` object literal with a bunch of properties and methods!

Problem: You can't access it from outside the IIFE!

F) Now we're going to expose (or export) the necessary (public) methods of `sound` so that the rest of the app can access them.

Add the following to **sound.js**, at the bottom of the IIFE

```
return{
    init: init,
    stopBGAudio: stopBGAudio,
    playEffect: playEffect
};
```

This is called the *Revealing Module Pattern,* where we reveal public references to methods inside the `sound` module's scope.

2) To hook up this module to **main.js**, head over to **loader.js** and make

```
window.onload = function(){
    console.log("window.onload called");
    // This is the "sandbox" where we hook our modules up
    // so that we don't have any hard-coded dependencies in
    // the modules themselves
    // more full blown sandbox solutions are discussed here:
    // http://addyosmani.com/writing-modular-js/
    app.sound.init();
    app.main.sound = app.sound;
    app.main.init();
}
```

`window.onload` look like this:

Reload the HTML page to be sure there are no syntax errors.

3) Now let's use use the **sound.js** module to hear some audio.

A) In **main.js**, add a new property for our **sound.js** module:

```
sound : undefined, // required - loaded by main.js
```

Technically we don't need to declare this property, because it will be created automatically by **main.js** in `window.onload`.

Why are we doing this? Because when we come back to this code in 6 months we might forget that our code is depending on the **sound.js** module.

B) In both locations where you are calling:

```
 this.playEffect();
```

instead call:

```
this.sound.playEffect();
```

C) Reload the page and test it. You should still hear the effect sound.

4) Now let's get the **sound.js** module to play the background sound … oops … we never wrote a method or exposed a property to do so!

A) Head back to **sound.js** and add this function to the IIFE

```
function playBGAudio(){
          bgAudio.play();
     }
```

B) Export it with the others as a property of the object we are returning.

```
…
playBGAudio: playBGAudio,
…
```

C) Now head back to **main.js**. In the locations where you are calling

```
this.bgAudio.play();
```

instead call

```
this.sound.playBGAudio()
```

D) And make `main.stopBGAudio()` look like this:

```
stopBGAudio: function(){
     // this.bgAudio.pause(); // OLD
     // this.bgAudio.currentTime = 0; // OLD
     this.sound.stopBGAudio()
}
```

E) Reload and test - it should work as before

F) In **main.js**, comment out all of the old sound code (except the `sound` property obviously) - anything that refers to `bgAudio, effectAudio, currentEffect, currentDirection` and `effectSounds`

Reload the page - if everything works OK - go ahead and delete the code you commented out.

WOW - we're finally done with this.

Yes, that was a lot of refactoring - but hopefully you see the advantage of the module pattern - we got to move a bunch of code out our main module.

An even better plan would have been to make the sound module up front. I'm sure you'll do so on your own projects.

Hopefully you also see the advantage of planning ahead, so you can avoid some of the more painful refactoring we did here.

Part VII - Submission

- You'll need to finish up Boomshine on your own - see the next and final Boomshine assignment for details

- Look back over have we done and talked about in the Boomshine A-E - it's quite a lot!

You now have a complete game skeleton that you can use for your game project:

- drawing
- animation
- screens and game states
- scoring
- collision detection
- mouse hit testing
- keyboard shortcuts
- keyboard control
- sound
- pausing
- OOP (object literals)
- modules:
  - creating and exporting a public interface
  - loading
  - adding
  - calling methods of modules

Some other topics we'll get to in the next few days with our demos:

- drawing images and sprite sheets
- drawing a particle system
- JS function constructors

Boomshine E Rubric

DESCRIPTION	SCORE	VALUE %
Circle Color – Circle color works correctly and color stays consistent after clicking the circle.		**20**
Background Music – Background music plays correctly.		**10**
Background Pause – Background music pauses correctly.		**10**
Explosion Sound – Each explosion plays the correct sound.		**10**
Ascending Pitch – Pitch ascends correctly with explosions.		**10**
Module – Code correctly inside of a module.		**15**
P Button Toggle Pause – P button toggles pause correctly.		**10**
Cheat Code – Shift + up cheats and increases score correctly.		**15**
Buggy Collision – Circle collision is buggy in some way.		**-20%**
Errors Thrown – Any errors thrown in the console.		**-20% (this time)**
Previous Work – Deductions for any parts from previous assignments that no longer work correctly. There are no set values for penalties. The more penalties you make the more points you will lose.		
Additional Penalties – These are point deductions for missing submission links, poorly written code, etc. There are no set values for penalties. The more penalties you make the more points you will lose.		
TOTAL		**100%**

We're done with this for now - ZIP and Post! **Include a link to your work on Banjo in the submission comments.**