# Testing Report

## Module Testing

Document input, expected output, and actual output for ten modules  
■ These modules must be essential to the functionality of the program  
■ You will need multiple test cases to complete thorough testing  
■ If applicable, identify the cause of and solution to errors

## Program Testing

Test each feature of the program  
■ Document the functionality, usability and performance of each  
■ If applicable, identify the cause of and solution to errors

## System Testing

Test your software in as many different environments as possible.  
This could include different operating systems, different browsers,  
different hardware, etc.  
■ Document the functionality and performance of the software. This  
may include the use of screenshots, traces, or response times.

# Evaluation

○ A detailed description of the finished program, identifying what features were  
implemented and how well they work  
○ Identify any features that were not implemented and explain why they were  
not included  
○ Discuss any major issues that were encountered during development  
○ Describe how the project challenged and/or improved your programming  
ability throughout its development

# Logbook & Bibliography

## Log 1: Basics

### Ctrl + C; Ctrl + V

#### Context

After not doing anything for the majority of the holidays, I have finally forced myself to begin.

The easiest and arguably most beneficial thing I could do at the moment would be familiarising myself with [matter.js](https://brm.io/matter-js/), the physics engine to be used throughout the project.

#### Implementation

I had nothing better to do so I copied the example code off their ‘Getting Started’ page. The code created two rectangles falling down onto solid ground with gravity and collisions handled by the engine.

A screenshot of a video game

Description automatically generated

I have no idea why the viewport is tiny

The code started by assigning aliases to modules, which I’d assume makes them important.

A screen shot of a computer

Description automatically generated

From what I can work out:

Engine includes the world in which the physics of bodies are calculated in, and some of the fundamental physics attributes (e.g. gravity)

Render is the canvas of matter.js, including methods and properties relating to rendering the objects in the world (e.g. how often objects are redrawn, displaying performance information)

Runner is a substitute for the normal main loop of a program, recalculating the physics of all objects in the world at set intervals. Alternatively, Engine.update calculates the physics of all objects a single time, allowing for more precise control over updating the physics simulation.

Bodies contains methods of creating common bodies to be simulated (e.g. rectangles, circles). Body, on the other hand, not only is able to create more complex bodies by inputting vertices, but also allows for manipulation of physical properties (e.g. angular velocity, friction, time scaling).

Composite is used for creating containers to sort bodies, somewhat like the usage of layers in digital art. The root world in which bodies are placed, engine.world is also a Composite.

#### Sources

##### [Matter.js source code](https://github.com/liabru/matter-js/blob/master/build/matter.js)

Downloaded the file to use the physics engine.

* By [liabru](https://github.com/liabru)
* Under MIT licence, which allows modification, distribution and private use so long as the MIT licence is still included in the software. Licence conditions are met as the MIT licence takes up the first 27 lines of the matter.js file.
* Accessed 21/01/2024

##### [Matter.js getting started page](https://github.com/liabru/matter-js/wiki/Getting-started)

Followed instructions to install matter.js and copied the code for learning purposes.

* By [liabru](https://github.com/liabru)
* Accessed 21/01/2024

### Visuals

#### Context

After having a basic scene, I wanted to see the capabilities of the renderer and how I could include these visuals in a game.

#### Implementation

I started by changing the size of the canvas, since a platformer would probably need a full screen display.

Initially I tried putting width and height properties alongside the others that already exist in the given code for Render.create. That didn’t work since width and height properties are for some reason nested within another options record, so it ended up looking like this in code:

A screen shot of a computer code

Description automatically generated

I considered dynamically resizing the canvas to fit the size of the viewport as I did with the orbital simulation. It didn’t seem like a great idea however since pixel perfect precision would be required in a platformer, unlike an orbital simulation where most of the canvas is just empty space. These dimensions looked alright on the screen, so I kept it as so for the time being.

A screenshot of a video game

Description automatically generated

There’s meant to be a large white bar at the bottom of the screen though I don’t think it’s visible in the document because of the white background

Reading the documentation, it turns out that matter.js also has a lot of debug options in its renderer, so I decided to turn all of them on while adding a few more bodies (I have no idea what any of them actually do though):

A screenshot of a computer game

Description automatically generated

It’s also possible to give fill colour to bodies once wireframe is disabled. By default, matter.js assigns each body a random colour chosen from a pre-written set. I also changed the background colour along the way.

A screenshot of a computer

Description automatically generated

There’s also a method called setPixelRatio for Render. The idea is that pixels specified in CSS can be scaled on the user’s screen. Normally, zooming in on elements would result in blurriness due to a single CSS pixel taking up multiple pixels on the user’s screen. However, if the pixel ratio is adjusted accordingly, the element will maintain its resolution despite zooming in or out.

That’s under ideal conditions anyhow. As of now I’m not sure how the pixel ratio will handle fractional pixels and whether or not it will affect the positioning of elements on screen. Quite important for the placement of platforms and items.

#### Sources

##### [Matter.js documentation – Matter.Render](https://brm.io/matter-js/docs/classes/Render.html)

Used for understanding render options.

* By [liabru](https://github.com/liabru)
* Accessed 23/01/2024

##### [MDN web docs - Window: devicePixelRatio property](https://developer.mozilla.org/en-US/docs/Web/API/Window/devicePixelRatio)

A page linked to by the matter.js documentation to explain the pixel ratio.

* By Mozilla
* Accessed 23/01/2024

### Keyboard Controls and Moving

#### Context

As per the platformer genre, the player should be able to move an in-game character through some sort of keyboard interaction – in this case the arrow keys. This will hopefully be a rough sketch of how basic keyboard controls should work.

#### Implementation

Writing it in code should be simple enough, just add event listeners for each key and use the applyForce method to move the object.

That was not the case. Turns out, JavaScript event listeners for the keydown event don’t detect the key being held down properly. When a key is held down, instead of the event being constantly fired, it fires once, then pauses for a while, before it finally starts firing constantly.

I’ve seen the exact same problem in graphics.py, and I assume it occurs due to the code not actually reading keystrokes but the text that the keys type out. Due to how typing in a text document works, holding down a key types a single letter first, pauses, and then spams the letter out (try typing ‘aaaaaaa’ by holding down ‘a’ and you’ll see what I mean). While this makes sense in typing, it does not make sense in a game.

This also results in some weird behaviour when multiple keys are pressed. Only if keys are pressed simultaneously (on the same frame I’d assume) do they both register as inputs, and the pausing midway problem still persists. On the other hand, if one key is pressed later than another, the later key will override the previous key. You really don’t think much into how differently the keyboard is used in gaming and office environments until this happens.

I considered using both keydown and keyup events to precisely track when a key is pressed and when it is released with flags, but felt it may be too complex for a simple problem, and decided to have a look at what the internet says.

As it turns out, this problem is stupidly common to the point that people have made entire libraries dedicated to fixing how JavaScript responds to holding down keys. While I could import perhaps [p5](https://p5js.org/reference/#/p5/keyIsDown) or [Keydrown](https://jeremyckahn.github.io/keydrown/) to fix this problem, it just doesn’t feel quite as fun as making it myself, and I have no idea how the 85% code originality works so I’ll just play it safe. Anyhow, the Stack Overflow responses gave similar approach as my initial solution so it should be fine.

And it was, surprisingly enough, fine.

#### Sources

##### [KeyDrown](https://jeremyckahn.github.io/keydrown/)

A library that just fixes the key hold problem. Had a look at it. Didn’t import it in the end.

* By [jeremyckahn](https://github.com/jeremyckahn)
* Accessed 24/01/2024

##### [p5 - keyIsDown](https://p5js.org/reference/#/p5/keyIsDown)

A rendering library that also fixes the key hold problem. Looked at it. Didn’t import.

* By [Processing Foundation](https://processingfoundation.org/)
* Accessed 24/01/2024

##### [A Stack Overflow question about the key hold problem](https://stackoverflow.com/questions/29279805/keydown-doesnt-continuously-fire-when-pressed-and-hold)

Used to check if my logic was correct on solving the key hold problem

* Response from [Ghis](https://stackoverflow.com/users/1311952/ghis)
* Accessed 24/01/2024

## Log 2: Movement Physics

### Acceleration and Deceleration

#### Context

Generally, movement responding to a keystroke should occur in three stages:

1. Acceleration. This is when the character speeds up and starts moving after a movement key is pressed.
2. Maximum velocity. This is when the character is moving at full speed, and will remain to until it is forced to stop one way or another.
3. Deceleration. This is when the character slows down, eventually stopping after the movement key is pressed.

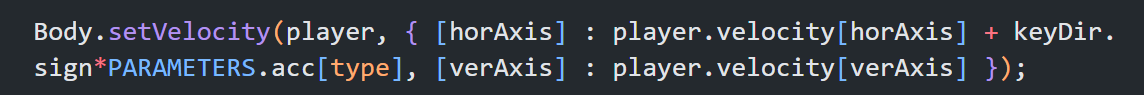
For any basic movement functionality, the above 3 would have to be implemented into the game.

#### Implementation

Breaking down the problem a bit more, I decided to use setVelocity rather than applyForce to accelerate and decelerate the box. This was mainly as the former was more general (suppose if I wanted to box to completely stop. I could just make velocity 0 immediately, or carefully apply the right amount of force for it to stop and not start moving in the opposite direction) and did not rely on the mass of the box.

I also considered using friction to implement a maximum velocity and decelerate the box, but it seemed too complex and depended on too many values. I ended up using setVelocity again for that. It seems now to be a common theme that matter.js is too accurate in terms of its physics for this game. Hopefully that doesn’t come back to bite me.

After my first attempted implementation of moving the box, it stood completely still… No errors in the console either. After a lot of head scratching, it turns out that .velocity is a read-only property? Not really sure why that’s the case. I can see myself having to write massive chunks of code just to move the box in the future.



Update from future me: I was more correct than I could’ve ever imagined

Also I realised that properties had to be changed through functions so that all other associated attributes would be changed accordingly and in order to ensure accuracy of simulation

Anyhow, on second attempt I actually got an error. Wohoo!

A screen shot of a computer error

Description automatically generated

The problem this time turns out to be that matter.js decided to namespace .setVelocity to be under Body rather than each instance of Body created. Another weird thing that I’m just not going to question about this physics engine.

Anyhow, that was the most interesting parts of implementing acceleration, just repeat that in every direction and now I’ve got move left/right, jump, and fast fall.

Deceleration is simpler in concept. If velocity is positive, decrease it. If velocity is negative, increase it. Toss that into a loop and it will keep repeating itself until eventually velocity approaches 0.

Similar idea for the velocity cap. If velocity is greater than positive cap, reset it to positive cap. If velocity is less than negative cap, reset it to negative cap.

#### Sources

##### [Matter.js documentation – setVelocity](https://brm.io/matter-js/docs/classes/Body.html#method_setVelocity)

Used to figure out how to change the velocity of the box.

* By [liabru](https://github.com/liabru)
* Accessed 28/01/2024

### Collision

#### Context

Now onto one of the biggest reasons I decided to use someone else’s physics engine – I didn’t want to deal with collisions.

While I did write up a pseudocode algorithm for it in planning, it is stupidly long even in pseudocode, and there’s no guarantee it’ll work.

Anyhow, right now I need to check if the box is on ground so this platformer game doesn’t turn into flappy bird.

#### Implementation

On the bright side, without writing any code at all myself, the box knows to collide on ground and not fall into inexistence.

On the not so bright side, Collsion.collides in matter.js is pretty stupid. It functions practically the same as my pseudocode in that it can only check for collisions between two given objects at a time. The function will return either null if no collision occurs, or an illegible collision object, thankfully pretty easy to put into an if statement.

A screen shot of a computer code

Description automatically generated

Yes I understand what this means

After putting it into an if statement, check if the collision record is null, all that was left was to setting a flag to tell if the box is on the ground, and put the whole thing into a loop to check for every single obstacle in existence.

A screen shot of a computer code

Description automatically generated

### Wall Jump

#### Context

#### Implementation

How do you prevent the player from just scaling the wall

Cooldown timer?

Just make the player bounce off the wall

How do you make sure it bounces off in the right direction instead of going into the wall again allowing for another wall jump and then just scaling the wall again

Make grounds grounds make walls walls

Stupid thing started walljumping left on a wall to the left on the player turning it into scaling the wall again

Put only = forgor == why does js not give an error for that

## Log 3: Gravity💀

### Just changing gravity

#### Implementation

Matter.js has this thing called engine.gravity that changes the x-y vector of gravity.

Add key detection to that and we’re done – simple enough

#### Sources

### Changing keystrokes and directions with gravity

#### Context

Suppose gravity pulls you onto the right wall, and from your perspective the right wall is the ground.

If we assume that gravity always pulls down, we can say that pressing the right arrow (moving right on screen) will make you fall. Similarly, pressing the left arrow should make you jump.

Pressing the up arrow would result in you moving up on screen, but right from your point of view. Pressing the left arrow would result in moving down on screen, but left from your point of view.

Since the functions jump, fastFall and move are called depending on keystrokes, the program needs to translate the keys pressed to the actual directions in your perspective for the functions to be called correctly.

Furthermore, when “right” in your perspective is passed into move in this case (pulled onto the right wall), the program needs to know that “right” in this case is not increasing the x value, but actually decreasing the y value. So the values that are assigned to moving in each direction respectively will also have to be changed.

#### Implementation

Use a bunch of records to convert this stuff

Took a while to figure out the best way to structure records

A computer screen shot of a code

Description automatically generated

And then every single movement function had to be changed so that movements could be carried out on any axis. This was made a bit more annoying as matter.js has read only properties, so I would have to assign values to a record every time to move.

Initially I resorted to a lot of if statements to check which axis the movement was meant to occur in, but I realised later that thankfully, records don’t care about the order of items, so I could do something like this: (also apparently adding square brackets just turns a string into an expression?)

A screen shot of a computer

Description automatically generated

While it is nearly impossible to read, I can assure that it does in fact work. To be fair, the if statement approach worked too, just that it took up twice the amount of lines.

#### Sources

### Changing detection with gravity

#### Context

Initially I wanted to just start on the next heading down there, but I realised that no matter how hard I tried, I could not get the block to jump when gravity was flipped in any direction other than downwards.

After a lot of console logging, the error was that the jump function was called at all.

After a bit more console logging, and realising that one of my flags couldn’t be accessed directly from terminal (why?), it was because the block didn’t actually think it was on the ground when gravity was reversed, and refuses to jump in mid-air.

It also resulted in weird stuff like it trying to fall even though it was already on the ground.

So, the thing I had to do first was make it realise it was colliding with the ground

#### Implementation

Obstacles of each direction are stored in arrays. The arrays are then put into a record, with the key of the record being the four directions.

Initially, I just copied the directions of the KEYSTROKE\_TO\_DIRECTION record which changed depending on gravity. While it was perfectly functional for upwards and downwards gravity, it was different for left and rightwards gravity.

e.g. when gravity to the left, pressing up would result in moving left from block’s perspective. However, obstacles to the right of the block from its perspective, would actually be rightwards on screen (I had to read this 4 times to make sure it made sense, not sure if it does now)

A screen shot of a computer program

Description automatically generated

### Changing physics with gravity

#### Context

Now we can finally do this.

#### Implementation

Since DIRECTION\_TO\_VALUE had a sign component already, all that was left was the multiply it to every movement.

And now that the block actually thinks it’s on the ground, ground friction and jumping finally work now.

Not much to say, just a lot of replacing stuff.

#### Sources

### Temporary Gravity Change & Sticking onto Walls

#### Context

While planning a bit ahead for level design, I realised it practically impossible to design interesting levels if the player has the ability to effectively teleport onto the other side of the screen, so it made sense to rework the core gravity mechanics of the game.

I eventually settled on the gravity change in all directions being temporary – that gravity would revert back to the previous direction after a set amount of time unless the player hits a wall in the direction gravity is changed, in which case the player would stick to the wall (gravity being in direction of wall until next change).

**Write something about sticking to while falling in opp dir**

#### Implementation

#### Sources

### Reworking Collision Detection [nested]

#### Context

The problem with the current method of collision detection became noticeable for detecting collision after gravity got reverted.

So, if the player only sticked onto walls before reverting gravity, it would often result in weird cases where (taking original direction as downwards and changing gravity upwards to stick to ceiling as an example):

* The block is still moving upwards
* But gravity has reverted to downwards
  + i.e. the block is moving upwards due to inertia rather than gravity
* The block clearly hits the ceiling but doesn’t stick to it
* And then falls all the way back down

This makes the game mechanic way too unintuitive, and creates more frustration rather than difficulty in the game, making it poor game design.

While the direction of gravity is always considered relatively downwards in previous code, this becomes problematic after gravity is reverted, in which case relative direction is difficult, if not impossible to determine.

I was too lazy to think so I just added extra code to detect collision in absolute directions.

#### Implementation

#### Sources

### Reworking Deceleration

#### Context

After playing around with the movement for a while, I felt like there was too much stopping force on the player. While high deceleration made stopping player movement crisp, it also made starting player movement a pain because deceleration was applied every tick and would cancel out player inputs to move.

While staring at the functions, I also realised that there wasn’t a velocity cap for the vertical axis this whole time – which very much explained why jumping and then changing gravity could make the player go on the ceiling as easily as it does.

#### Implementation

#### Sources

### Key + Direction for Gravity Change

#### Context

I felt from all this testing that pressing WASD while using arrow keys wasn’t most comfortable way to control the character. This was similarly echoed from feedback I’ve gotten showing this to others, the most common of which being people spamming the space bar for gravity change.

#### Implementation

#### Sources

### Wavedashing!

#### Context

While there is already the movement option of changing gravity, there is a need for some more advanced/obscure movement techniques. I took *a* *bit* of inspiration from Celeste wavedashing, where if the player dashes into the ground and jumps immediately after, dash momentum is inherited and players can jump long distances.

I’d like to do something similar in this game. The idea is after landing from gravity change (within a set period of time), the player can jump at a higher velocity than usual.

#### Implementation

In code terms, this means to implement a timer for how long the player is on ground. The timer is incremented while they are on ground and reset once they leave the ground. While the timer is less than a given value, velocity will not be capped.

I tried messing around with some values of the ground timer, but got mixed up (a lot) with inequality signs. Eventually, I just resorted to setting an initial value, and then decrementing it gradually on ground. While the ground timer is less than 0, it will cap velocity; while the ground timer is greater than zero, it will not cap velocity.

That ran into another problem, which is that velocity immediately gets reset after jumping, as the timer gets reset once the box is not on ground. The other problem this was associated with was that the momentum boost happened every time the box jumped and landed back on ground, rather being triggered by a gravity change.

After reading through the code for a while, I initialised the ground timer within the \_gravReverted section of changing gravity, and the timer would decrement itself both on ground and in air.

This made it so that the box had to fall back onto the same direction after gravity is changed, which was not the biggest concern. It also made the removal of the velocity cap not last the entire duration of the jump, as the timer would deplete before the jump ended. This was an unexpectedly balanced feature – the earlier the player jumped off the ground, the longer the would retain the uncapped velocity, awarding more precise inputs.

#### Sources

### Restricting Jump + Gravity Change

#### Context

One of the most persistent problems with the movement so far is with gravity change upwards. If a gravity change upwards is inputted shortly after a jump, the high starting velocity of the jump is inherited even after gravity changes, resulting in a stupidly high jump.

A screenshot of a computer

Description automatically generated

Being able to jump at the height of 70% of the screen is not balanced game design.

While the game is designed to award players performing precise inputs, this mechanic makes future level design quite difficult given that it allows the player to bypass too many obstacles. Some methods of fixing this are as follows:

The most obvious solution to fixing this would be to make the velocity fixed during gravity changes. However, I find that gravity change would be mostly identical to a dash mechanic, and the sunk cost of having messed with gravity for so long doesn’t exactly allow me to just revamp the core movement mechanic. Additionally, I would like there to be some variation of how gravity change would work depending on the velocity before entering the gravity change (i.e. being able to vary how far the dash travels).

Another method would be to impose a maximum velocity on the upwards gravity change specifically. While this method contradicts less with the game’s intention than the former, implementing another maximum velocity function takes a fair bit of time and effort, as it is called exclusively during upward gravity changes, which would require for the detection of velocity in a relative direction (taking the direction of gravity as always being downwards). It could also cause some weirdness with visuals, as the box may immediately reach the velocity cap with no gradual acceleration, making it feel inconsistent with gravity changes of other directions.

The method I eventually settled was using another timer to ensure that gravity changes could only be triggered after being airborne for a set amount of time. This minimises the inconsistencies across gravity change directions, as all gravity changes can only occur after the set timer, and the player still has significant control over the speed and distance of gravity changes.

#### Implementation

Similar to the previously implemented gravTimer and groundTimer, airTimer would be incremented and reset depending on a flag – in this case \_onGround. Added checks of whether the box was on ground and if the timer was above a set value was enough to implement the fix.



Additionally, I used a lower opacity version of the default matter yellow to indicate when the player regained their gravity change but would not be able to do so as they are still on ground.

A screenshot of a computer

Description automatically generated

#### Sources

## Log 4: Resizing the Canvas

### Desperately Trying to Avoid Relative Coordinates

#### Context

Resizing the canvas isn’t hard in the slightest, just change the width and height of the canvas and it’s done.

The slightly trickier part is making sure the dimensions of objects are preserved and display identically across different resolutions. Ideally, I would like this platformer to be pixel perfect. This means that the canvas can’t just fill up the entire viewport, as someone playing the game on a Samsung smart fridge would see things very differently compared someone playing on a 4k TV.

Usually, this is solved by scaling both height and width by the same factor until one of the horizontal/vertical edges are reached, and using black bars to fill in the remainder of space.

The idea is to calculate the ratio of height to height and width to width separately.

Suppose the intended resolution is 1280 by 720 (16:9), but the actual screen is 800 by 600 (4:3).

The ratio of intended resolution to actual resolution is 1280/800 = 1.6 in width, and 720/600 = 1.2 in height. This means that each actual pixel on screen would represent 1.6 intended pixels in width, and 1.2 intended pixels in height. It can be seen that the width ratio is larger than the height ratio, which means each actual width pixel is containing more intended pixels than that of the height, and as the width is less than the height, the image becomes stretched vertically.

However, if we take the greater of the two ratios – the width ratio – and force the width to also use the same ratio, we can get a smaller image with a consistent ratio to fit in the larger actual screen, albeit with some black bars. That is, instead of the width being the full 600 pixel height on the actual screen, it turns into 720/1.6 = 450 pixels. So we have a 800 by 450 image within the 800 by 600 screen, with black bars of (600 – 450)/2 = 125 pixels height on the top and bottom.

*What could possibly go wrong?*

#### Implementation

Despite that lengthy explanation of how black bars work, the implementation isn’t all that hard, just consisting of calculating ratios, comparing ratios, and dividing the ratio back onto the other is shown.

The difficult part was more of me being stupid. Originally trying to make the game absolutely pixel perfect, I decided to floor the ratios to make sure there would be no weird fractional pixel values going on. That resulted in the canvas just straight up disappearing from the screen for lower zoom levels, as the ratio would be less than 1 and get floored to 0, causing a divide by 0 error.

I somehow didn’t realise that for a solid half hour and continued trying to debug it in the console. The funny thing was that opening the console made the window size smaller, acting equivalent as increasing the zoom level, which actually made the canvas appear. So, I spent a good while getting confused over why my code only worked when I was checking that it worked, but immediately stopped working whenever I just look away from the values. 💀💀💀

A screenshot of a computer

Description automatically generated

“Hmm, why is this only working when the console is open?”

Anyhow, the canvas would actually fit the screen after that, but the problem was everything in the world didn’t move, so half of everything just wasn’t visible.

A screenshot of a computer

Description automatically generated

While the obvious solution was to make everything relatively positioned, to convert coordinates based on the scaling of the canvas, which I actually knew how to do in the orbital simulation, I decided it was too prone to messing up from zoom levels and not being pixel perfect, and jumped down a rabbit hole of many methods which didn’t work.

While there was the slightest bit of useful info, like how the pixel ratio in matter.js doesn’t actually change the coordinates of things in the world, but just makes images look crisper with higher resolution, which I spent half an hour messing with and then the documentation said you could just set it to ‘auto’ and it does the thing itself…

I also tried to mess with Composite.scale which supposedly scales everything within the composite. Not entirely sure if it had any impact on anything.

I think I’ll leave resizing the canvas to the end if I have time to even do it.

#### Sources

[Matter.js Documentation - pixelRatio](https://brm.io/matter-js/docs/classes/Render.html#property_options.pixelRatio)

For understanding how the pixel ratio affected objects in the world

* By [liabru](https://github.com/liabru)
* Accessed 6/05/2024

## Log 5: Platforms

### Just the Basics

#### Context

Now that movement is (mostly) complete, finally I would be forced to make a platform.

#### Implementation

Instead of having code lie around in blank space as I usually do, I decided to make the platforms a class. This would make life easier for having multiple platforms be on screen simultaneously and to modify platforms (e.g. making falling platforms and disappearing platforms, if I ever get to that. I’ll also how to figure out how superclasses and subclasses work in js). I probably should have made the player a class too, but I was too lazy to do that and since movement works there’s no point in fixing it.

The class creation isn’t all that interesting, create a default class with some parameters of a point and its dimensions, added it to the array of existing obstacles and put it into the world. The only slightly interesting thing worth mentioning is since that I made collision detection direction based, I would have to create four different faces for each platform such that movement mechanics worked properly.

A black rectangle with yellow dots and white lines

Description automatically generated

What the average platform looks like

#### Sources

[MDN Web Docs - Classes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)

For the syntax of creating classes

* By Mozilla
* Accessed 10/0502024

### Spikes

#### Context

Pretty much the same idea as the basic platform. I didn’t implement death yet so it shouldn’t be much different.

#### Implementation

The biggest difficulty of creating spikes would lie in its shape and tile-ability (I just made that word up).

Ideally, I’d like my spikes to be triangles. They could be a rectangle shaded in another colour which I call lava or something, but that doesn’t quite feel right, especially if I’m considering making them fall from the ceiling.

Creating the shape wasn’t too hard, as matter had already a method that created regular polygons. Changing the direction of the spikes was slightly harder, as it required trial and error to determine the correct angles, which could be summarised by this record:

A computer screen shot of a code

Description automatically generated

For the tile-ability, the idea is that each spike object should consist of many spikes, as it is unlikely for there to be a singular spike in any case. I did this in a for loop, where a coordinate is incremented a set amount as a spike is created for the coordinate, looping from a start coordinate to the end coordinate.

A group of triangles with numbers

Description automatically generated

### Death

#### Context

Despite having spikes, they do nothing right now. Even worse than platforms since I didn’t put in code for the game to think spikes were ground, and the player can’t even jump off them (maybe I should have a 2 frame window for players to be able to jump off spikes and not die?).

Anyhow, the point is that the player needs to die.

So, the goal is to have the box splatter into a million pieces upon coming in contact with spikes. Sadly, a million particles going around the screen tends to cause severe performance issues (visible frame drops when more than 100 particles in reality), so the number will have to be tuned down a bit.

#### Implementation

Ideally, I would just have to remove the box from the world, replace with a bunch of tiny boxes cramped into the same location, and the physics engine of matter.js should have the clip into each other and explode for me.

Sadly, matter.js was once again too realistic, and the boxes gained no velocity from cramming them against each other. Instead, I would have to apply my own velocity.

To make the death animation unique and intriguing each time, the particles should be launched at random velocities. So, I put in the velocity as Math.random() which generated a random float between 0 and 1, and then multiplied it by 10 to make the speed faster and more impactful.

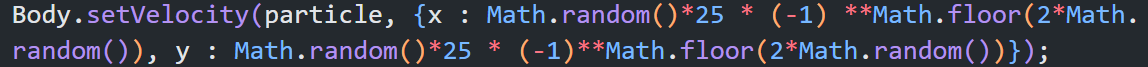
The problem with this method, after some testing, I realised was that the particles could only have positive velocity, i.e. they could only go rightwards and downwards. While I could fix this by writing up or even copying someone else’s separately written random function, I was too lazy and decided to work my own way through this.

Eventually I found this really smart method with indices of -1.

For even numbers , . For odd numbers , . This way, it was possible to generate the negative sign to multiply to values. Now all that was required to be able to generate a number that could be even or odd.

The floor function discards the digits behind the floating point, allowing for an integer to be obtained. While floor(Math.random()) would always return 0, as Math.random only generates numbers between 0 and 1 exclusive, floor(2\*Math.random()) would return either 1 or 0, allowing for a random odd or even integer, which matches the requirements here.

In other words, the code ended up like this. Not the prettiest nor most readable, but very compact.



Yes I do have line wrap on by default

After being able to splatter randomly, I also added in some time scaling to make the animation look nicer, and more impactful at the moment of death. I was going to originally add a freeze frame of some sort, but I couldn’t figure out how to use setTimeout in combination with matter.js so I ditched that idea.

I should probably figure out how to add screen shake to make the splattering feel more impactful

### Respawn

#### Context

As a believer in the circle of life, every death is followed by the birth; the player should respawn after they die, and carry out their normal platforming routine before reliving the traumatic experiences of death again, never being able to escape the Sisyphean cycle.

#### Implementation

Since setTimeout didn’t work properly with matter.js, I implemented my own timer by in game updates, using it to check for the duration passed and therefore being able to respawn after a set amount of time.

The actual respawning was pretty much death in reverse – remove all the particles from the world, put the player back into the world, and also teleport them to a pre-determined respawn point

Might add a quick respawn feature which kills the player directly and respawns (maybe at a faster speed) later.

Also, to make the death animation look nicer, I increased the particle count to 50 and used matter.js default random colours.

A screenshot of a video game

Description automatically generated

## Log 6: Fixing Movement

### Gravity Change

#### Context

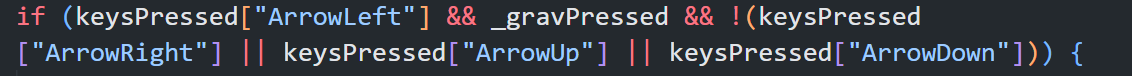
I am yet to implement diagonal gravity changes, and most likely won’t simply due to its chaotic nature.

However, keybinds associated with diagonal gravity change still works somewhat, just that the horizontal direction will be prioritised above vertical directions if both horizontal and vertical keys are pressed. From a bit of others playtesting, they seem to have all raised issues with that system, which is why it is being removed now.

#### Implementation

The hardest part was honestly digging through my spaghetti code to find the if statements for gravity change.

Afterwards, it was just adding a lengthy clause of checking no other arrows are being pressed before changing gravity.



### Non-Bufferable Wavedashes

#### Context

An unintended feature when implementing wavedashes was that the player could hold the up key in mid-air and after landing to guarantee a wavedash with full momentum. This does go against wavedashing being an advanced mechanic that requires precise inputs. As such, some way of removing previous inputs was required upon falling back on ground after a gravity change.

#### Implementation

My first and most obvious attempt was to run neutral to force all inputs be cancelled upon coming in contact with the ground. However, due to how the if statements were structured, wavedashing was made impossible as neutral was called continuously by the game loop until the wavedash timeframe had ended.

In other words, another \_waveDash flag would need to be added on top of the current groundTimer to only cancel inputs the first time the box touches ground, enabling the player to move later and then wavedash.

However, as neutral would cancel inputs in all directions, this meant that the on top of not being able to jump/wavedash, the player could also not move left nor right. This was inconsistent with landing normally from just a jump, and ideally should be avoided to make movement more consistent.

So, instead of cancelling all inputs, I decided check directly for the keystroke for the upwards direction (which changes depending on gravity), not cancelling it but only using it to deactivate the \_waveDash flag.

Despite so, wavedashing was still bufferable. After half an hour of removing and readding unnecessary flags, in just turned out that I forgot to reintroduce the velocity cap when the \_waveDash flag was false.

### Title

#### Context

#### Implementation

#### Sources

○ Regular, detailed and reflective entries. Each entry may include:  
■ Description of what was attempted  
■ Discussion of problems and possible solutions  
■ Evaluation of progress and/or feature/program  
■ Ideas and considerations  
○ Screenshots must be taken of the program at regular intervals to show the  
development of the software

○ For all external resources used (including images, code snippets, etc),  
identify:  
■ The original author (may be a username)  
■ Where the resource was accessed (provide a link if applicable)  
■ When the resource was accessed  
■ Any licence requirements of the resource  
■ How the resource was used  
○ Note that links must be specific, e.g., to a discussion page on Stack  
Overflow, not to the Stack Overflow home page