# Testing Report

## Module Testing

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
|  |  |  |
|  |  |  |
|  |  |  |

### move(dir, type)

By accessing the global player object, moves it in the given direction (left/right) by adjusting its velocity. The adjustment varies on the type given (horizontal/air).

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| dir = “right”  type = “hor”  PARAMETERS.acc.hor = 0.5  PARAMETERS.acc.air = 1  player.velocity =  {x:0, y:0} | player.velocity =  {x:0.5, y:0} | As expected |
| dir = “left”  type = “air”  PARAMETERS.acc.hor = 3.14  PARAMETERS.acc.air = -3.14  player.velocity =  {x:3, y:15} | player.velocity =  {x:-0.14, y:15} | Minor precision issues due to binary conversion, but still as expected |
| dir = “left”  type = “air”  PARAMETERS.acc.hor = 69  PARAMETERS.acc.air = 0  player.velocity =  {x:-5, y:-4.323556621234} | player.velocity =  {x:-5, y:-4.323556621234} | As expected |

### jump()

Using the player object, the direction of gravity and parameters, modifies the player’s velocity for it to jump in which ever direction may be upwards at the time (the upwards direction is always the direction against the pull of gravity).

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| gravDir = “down”  PARAMETERS.acc.jump = 7  player.velocity =  {x:0, y:0} | player.velocity =  {x:0, y:-7} | As expected |
| gravDir = “left”  PARAMETERS.acc.jump = 100  player.velocity =  {x:-100, y:-100} | player.velocity =  {x:0, y:-100} | Items in a different order but still as expected |
| gravDir = “up”  PARAMETERS.acc.jump = 1.41  player.velocity =  {x:1.89, y:-3390} | player.velocity =  {x:1.89, y:-3388.59} | As expected |
| gravDir = “right”  PARAMETERS.acc.jump = 1  player.velocity =  {x:90, y:0} | player.velocity =  {x:89, y:0} | Items in a different order but still as expected |

## maxVel(type)

Accesses the player object, ensuring its velocity does not exceed the maximum velocity by resetting values in the velocity record. The vertical velocity will have the same cap whereas the horizontal velocity has different caps depending on the type (i.e. on ground or in air) passed in.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| type = “air”  PARAMETERS.max.hor = 3  PARAMETERS.max.air = 6  PARAMETERS.max.ver = 15  player.velocity =  {x:0, y:0} | player.velocity =  {x:0, y:0} | As expected |
| type = “air”  PARAMETERS.max.hor = 3  PARAMETERS.max.air = 6  PARAMETERS.max.ver = 15  player.velocity =  {x:20, y:20} | player.velocity =  {x:6, y:15} | As expected |
| type = “hor”  PARAMETERS.max.hor = 100  PARAMETERS.max.air = 90  PARAMETERS.max.ver = 0  player.velocity =  {x:-200, y:-20} | player.velocity =  {x:-90, y:0} | -0 is a thing apparently? Numerical values are still as expected |
| type = “hor”  PARAMETERS.max.hor = 0.00001  PARAMETERS.max.air = 0.5  PARAMETERS.max.ver = 1.41  player.velocity =  {x:-3.14, y:-2.71 } | player.velocity =  {x:-0.00001, y:-1.41} | As expected |

## decel(type)

Reduces the horizontal component of the player object’s speed by a set value which varies depending on the type (i.e. on ground or in air) passed in. When the player has low enough speed horizontally (between -0.5 and 0.5 inclusive), completely cancels horizontal movement.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| type = “air”  PARAMETERS.dec.hor = 0.4  PARAMETERS.dec.air = 0.5  player.velocity =  {x:0, y:0} | player.velocity =  {x:0, y:0} | As expected |
| type = “air”  PARAMETERS.dec.hor = 0.4  PARAMETERS.dec.air = 0.5  player.velocity =  {x:12, y:50} | player.velocity =  {x:11.5, y:50} | As expected |
| type = “hor”  PARAMETERS.dec.hor = 1  PARAMETERS.dec.air = 2  player.velocity =  {x:-69, y:-420} | player.velocity =  {x:-68, y:-420} | As expected |
| type = “hor”  PARAMETERS.dec.hor = 1  PARAMETERS.dec.air = 2  player.velocity =  {x:0.5, y:-420} | player.velocity =  {x:0, y:-420} | As expected |

### respawn(spawnpoint, defaultGravDir)

Moves the player object to the spawnpoint, changing the flag indicating the player is alive, resetting the direction of gravity and all keystrokes.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| spawnpoint = {x:640, y:100}  defaultGravDir = “down”  gravDir = “down”  player.position =  {x:0, y:0}  directionsPressed = {  up: false,  down: false,  left: false,  right: false  } | player.position =  {x:640, y:100}  \_isAlive = true  gravDir = “down”  directionsPressed = {  up: false,  down: false,  left: false,  right: false  } | As expected |
| spawnpoint = {x:69, y:420}  defaultGravDir = “up”  gravDir = “down”  player.position =  {x:640, y:100}  directionsPressed = {  up: true,  down:true,  left: true,  right: true  } | player.position =  {x:69, y:420}  \_isAlive = true  gravDir = “up”  directionsPressed = {  up: false,  down: false,  left: false,  right: false  } | As expected |
| spawnpoint =  {x:-3.14, y:-4.13}  defaultGravDir = “left”  gravDir = “right”  player.position =  {x:0.001, y:53342}  directionsPressed = {  up: false,  down:true,  left: true,  right: false  } | player.position =  {x:-3.14, y:-4.13}  \_isAlive = true  gravDir = “left”  directionsPressed = {  up: false,  down: false,  left: false,  right: false  } | As expected |

### changeGrav(dir)

Changes the direction of gravity. This includes the variable storing the direction of gravity, changing the record containing the relative directions of walls to the up/down/left/right on screen, the relative directions which pressing arrow keys will now move the player object, and the relative horizontal and vertical axes and their corresponding signs. (That is a mouthful to both read and write, you don’t really need to understand what that means anyhow)

UPOBST, DOWNOBST, LEFTOBST, RIGHTOBST are replaced with stub strings rather than their actual arrays to show the changing of obstacles.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| dir = “up”  gravDir = “down”  engine.gravity = {x:0, y:1}  DIRECTION\_TO\_VALUE = {  up: { axis:"y", sign: -1 },  down: { axis: "y", sign: 1 },  left: { axis: "x", sign: -1},  right: { axis: "x", sign: 1 }}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "up",  ArrowDown: "down",  ArrowLeft: "left",  ArrowRight: "right",}  OBSTACLES = {  up: UPOBST,  down: DOWNOBST,  left: LEFTOBST,  right: RIGHTOBST,} | gravDir = “up”  engine.gravity = {x:0, y:-1}  DIRECTION\_TO\_VALUE = {  up: { axis: "y", sign: 1 },  down: { axis: "y", sign: -1 },  left: { axis: "x", sign: -1 },  right: { axis: "x", sign: 1 }}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "down",  ArrowDown: "up",  ArrowLeft: "left",  ArrowRight: "right",}  OBSTACLES = {  up: DOWNOBST,  down: UPOBST,  left: LEFTOBST,  right: RIGHTOBST,} | Some record items are out of order, but do not affect accessing items; as expected |
| dir = “left”  gravDir = “down”  engine.gravity = {x:0, y:1}  DIRECTION\_TO\_VALUE = {  up: { axis:"y", sign: -1 },  down: { axis: "y", sign: 1 },  left: { axis: "x", sign: -1},  right: { axis: "x", sign: 1 }}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "up",  ArrowDown: "down",  ArrowLeft: "left",  ArrowRight: "right",}  OBSTACLES = {  up: UPOBST,  down: DOWNOBST,  left: LEFTOBST,  right: RIGHTOBST,} | gravDir = “left”  engine.gravity = {x:-1, y:0}  DIRECTION\_TO\_VALUE = {  up: { axis: "x", sign: 1 },  down: { axis: "x", sign: -1 },  left: { axis: "y", sign: -1 },  right: { axis: "y", sign: 1 },}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "left",  ArrowDown: "right",  ArrowLeft: "down",  ArrowRight: "up",}  OBSTACLES = {  up: "RIGHTOBST",  down: "LEFTOBST",  left: "UPOBST",  right: "DOWNOBST",} | Some record items are out of order, but do not affect accessing items; as expected |
| dir = “right”  gravDir = “left”  engine.gravity = {x:-1, y:0}  DIRECTION\_TO\_VALUE = {  up: { axis: "x", sign: 1 },  down: { axis: "x", sign: -1 },  left: { axis: "y", sign: -1 },  right: { axis: "y", sign: 1 },}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "left",  ArrowDown: "right",  ArrowLeft: "down",  ArrowRight: "up",}  OBSTACLES = {  up: "RIGHTOBST",  down: "LEFTOBST",  left: "UPOBST",  right: "DOWNOBST",} | gravDir = “right”  engine.gravity = {x:1, y:0}  DIRECTION\_TO\_VALUE = {  up: { axis: "x", sign: -1 },  down: { axis: "x", sign: 1 },  left: { axis: "y", sign: 1 },  right: { axis: "y", sign: -1 },}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "right",  ArrowDown: "left",  ArrowLeft: "up",  ArrowRight: "down",}  OBSTACLES = {  up: "LEFTOBST",  down: "RIGHTOBST",  left: "DOWNOBST",  right: "UPOBST",} | Some record items are out of order, but do not affect accessing items; as expected |
| dir = “down”  gravDir = “right”  engine.gravity = {x:1, y:0}  DIRECTION\_TO\_VALUE = {  up: { axis: "x", sign: -1 },  down: { axis: "x", sign: 1 },  left: { axis: "y", sign: 1 },  right: { axis: "y", sign: -1 },}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "right",  ArrowDown: "left",  ArrowLeft: "up",  ArrowRight: "down",}  OBSTACLES = {  up: "LEFTOBST",  down: "RIGHTOBST",  left: "DOWNOBST",  right: "UPOBST",} | gravDir = “down”  engine.gravity = {x:0, y:1}  DIRECTION\_TO\_VALUE = {  up: { axis:"y", sign: -1 },  down: { axis: "y", sign: 1 },  left: { axis: "x", sign: -1},  right: { axis: "x", sign: 1 }}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "up",  ArrowDown: "down",  ArrowLeft: "left",  ArrowRight: "right",}  OBSTACLES = {  up: UPOBST,  down: DOWNOBST,  left: LEFTOBST,  right: RIGHTOBST,} | Some record items are out of order, but do not affect accessing items; as expected |
| Dir = “down”  gravDir = “down”  engine.gravity = {x:0, y:1}  DIRECTION\_TO\_VALUE = {  up: { axis:"y", sign: -1 },  down: { axis: "y", sign: 1 },  left: { axis: "x", sign: -1},  right: { axis: "x", sign: 1 }}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "up",  ArrowDown: "down",  ArrowLeft: "left",  ArrowRight: "right",}  OBSTACLES = {  up: UPOBST,  down: DOWNOBST,  left: LEFTOBST,  right: RIGHTOBST,} | gravDir = “down”  engine.gravity = {x:0, y:1}  DIRECTION\_TO\_VALUE = {  up: { axis:"y", sign: -1 },  down: { axis: "y", sign: 1 },  left: { axis: "x", sign: -1},  right: { axis: "x", sign: 1 }}  KEYSTROKE\_TO\_DIRECTION = {  ArrowUp: "up",  ArrowDown: "down",  ArrowLeft: "left",  ArrowRight: "right",}  OBSTACLES = {  up: UPOBST,  down: DOWNOBST,  left: LEFTOBST,  right: RIGHTOBST,} | Some record items are out of order, but do not affect accessing items; as expected |

### checkLevelComplete(), requires Transitioner(centreX, centreY, width, height)

Given an array of end records, each with a transitioner indicating the area in which the player object must be for the level to be considered complete, determine if the player has reached any of these ends and return the end object if applicable.

Note also the player is 40 by 40.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| player.position = {x:0, y:0}  level.ends = [  { range:  Transitioner(0, 0, 100, 100), next: 1 },  ] | {  range:  Transitioner(0, 0, 100, 100), next: 1  } | As expected |
| player.position = {x:100, y:0}  level.ends = [  { range:  Transitioner(0, 0, 100, 100), next: 1 },  ] | null | As expected |
| player.position = {x:520, y:90}  level.ends = [  { range:  Transitioner(0, 0, 100, 100), next: 1 },  { range:  Transitioner(500, 50, 100, 100), next: 2 },  ] | { range:  Transitioner(500, 50, 100, 100), next: 2 } | As expected |
| player.position = {x:-69, y:90}  level.ends = [  { range:  Transitioner(0, 0, 100, 100), next: 1 },  { range:  Transitioner(500, 50, 100, 100), next: 2 },  ] | null | As expected |

### Permanent Gravity Change First Half (not organised into a function in the code)

Changes gravity permanently in the current direction (changes the revert direction to be the same as the current direction of gravity) if the player object is touching a wall in the direction (given that gravity pulls downwards at all times, the player object is touching the ground from its perspective).

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| \_onGround = true  \_gravChanged = true  gravDir = “up”  gravRevert = “down” | \_gravChanged = false  gravRevert = “up” | As expected |
| \_onGround = false  \_gravChanged = false  gravDir = “left”  gravRevert = “left” | \_gravChanged = false  gravRevert = “left” | As expected |
| \_onGround = false  \_gravChanged = true  gravDir = “right”  gravRevert = “down” | \_gravChanged = false  gravRevert = “down” | As expected |
| \_onGround = true  \_gravChanged = true  gravDir = “right”  gravRevert = “down” | \_gravChanged = false  gravRevert = “right” | As expected |

### isNeutral()

Returns true if no keystrokes are inputted, false otherwise.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| directionsPressed = {  up: false,  down: false,  left: false,  right: false}  gravPressed = {  W: false,  A: false,  S: false,  D: false} | true | As expected |
| directionsPressed = {  up: true,  down: true,  left: true,  right: true}  gravPressed = {  W: true,  A: true,  S: true,  D: true} | false | As expected |
| directionsPressed = {  up: false,  down: true,  left: false,  right: false}  gravPressed = {  W: false,  A: false,  S: false,  D: false} | false | As expected |
| directionsPressed = {  up: false,  down: false,  left: false,  right: false}  gravPressed = {  W: false,  A: false,  S: false,  D: true} | false | As expected |

### Platform.create(x, y ,width, height)

Creates a platform with centre (x, y), and dimensions of width and height.

The platform is formed of four individual rectangles for each of its faces, and is hollow inside.

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected** | **Output** |
| x = 100  y = 100  width = 100  height = 100 | A square platform centred at (100, 100) with side length 100 | As expected |
| x = 500  y = 314  width = 5  height = 500 | A platform centred at (500, 250) with width 5 and height 500, very long vertically | As expected |
| x = 100  y = 100  width = 1  height = 1 | A square platform centred at (100, 100) with side length 1, barely visible | This feels like a UFO sighting  As expected |

## 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.

### Surface Pro 4 i5-6300U CPU 2.4GHz 8GB RAM over 40 tabs open Windows 10

# 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

# Bibliography

If there is no description on license requirements, no license exists for the source.

##### [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

##### [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

##### [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

[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

[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

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

Used to figure out how to use a tracking camera.

* By [liabru](https://github.com/liabru)
* Accessed 22/05/2024

[Matter.js demo – views](https://brm.io/matter-js/demo/#views) ([source code](https://github.com/liabru/matter-js/blob/master/examples/views.js))

For understanding how to change bounds of the camera.

* By [liabru](https://github.com/liabru)
* Accessed 22/05/2024

Jarrett

For teaching me and writing the code for the level transition screen

* Accessed 1/06/2024

##### [Easing Functions Cheat Sheet - easeInOutSine](https://easings.net/" \l "easeInOutSine)

Copied entire function.

* By [ai](https://github.com/ai)
* GPL-3.0 license. Permits all distribution and changes of the software for both public and private use. License conditions were not violated by copy pasting the code.
* Accessed 1/06/2024

[Matter.js demo – sensors](https://brm.io/matter-js/demo/#sensors) ([source code](https://github.com/liabru/matter-js/blob/master/examples/sensors.js))

For understanding how to use sensor bodies. Downloaded and modified for learning purposes.

* By [liabru](https://github.com/liabru)
* Accessed 07/06/2024

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

Used for reference in learning about the Sleeping module.

* By [liabru](https://github.com/liabru)
* Accessed 26/06/2024

##### [Matter.js Documentation – options.wireframes](https://brm.io/matter-js/docs/classes/Render.html" \l "property_options.wireframes)

Used as reference for wireframe rendering.

* By [liabru](https://github.com/liabru)
* Accessed 27/06/2024

##### [Matter.js Documentation – render.visible](https://brm.io/matter-js/docs/classes/Body.html" \l "property_render.visible)

Used as reference for wireframe rendering.

* By [liabru](https://github.com/liabru)
* Accessed 27/06/2024

# Logbook

## 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 most 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.

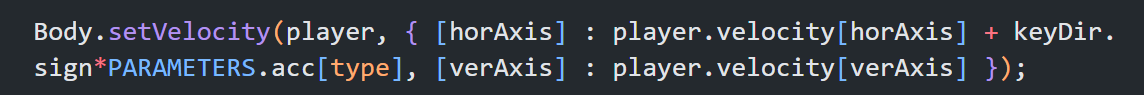
For any basic movement functionality, the above 3 would have to be implemented into the game.

#### 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 alternatively 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 (F=ma by Newton’s Second Law).

I also considered using friction to implement a maximum velocity and decelerate the box, but again 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 (I think that’s the terminology for that?) 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.

Movement is far from complete but I can at least by Flappy Bird with this now.

#### 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 stay as Flappy Bird with jumping in midair.

#### 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, definitely

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

This was one of the more niche and mostly unnecessary movement options for the game.

No real reason for implementing this, just felt like it would accompany the base movement set fairly well.

To also add some more difficulty to the game, wall jumps would be triggered by holding upwards and away from the wall while colliding with it, rather than the usual approach of just pressing jump while on a wall.

#### Implementation

The code was not hard; just copy the code for jump under a different keystroke combination and it’s done.

The bigger problem was balancing it the movement. If wall jump was the same as jump, there would be nothing preventing the player from continuously wall jumping and essentially scaling the wall.

I considered implementing some sort of cooldown timer for this, but decided it was not worth the effort, as it still would encourage wall scaling but waiting out the timer while on the wall. The simpler solution I settled on was making the player bounce of the wall in the opposite direction upon wall jump, directly preventing the player from quickly coming in contact with the wall again.

The problem now turned into how to know which direction the player should bounce off towards, even when gravity is changed all over the place. This took around a week of thinking to come up with a solution.

My (mostly inefficient) idea was to categorise each obstacle with a direction. This way, it would be known if for example the box collided with a left wall, and would be required to bounce rightwards. When gravity is changed towards another direction, the direction of obstacles would then be swapped around. For example if the direction of gravity was turned upwards from downwards, the left and right walls would not change, whereas the ceiling and floor would swap.

Afterwards was just a small error of the box wall jumping leftwards even though it was already on the left wall, which did not occur at all for the right wall. After some debugging it was because I put single equal signs (=) instead of double equal signs (==) for the if statement. 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.

### Changing keystrokes and directions with gravity

#### Context

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

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

Pressing the up arrow would result in you moving up on screen, but right from the relative perspective. Pressing the left arrow would result in moving down on screen, but left from the relative perspective

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 relative 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

With my best effort to avoid directly storing code and executing them with eval, I decided to structure them in records, which took a while to organise properly

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 and call a function every time for movement.

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 like eval?)

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 number of lines.

### 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 downwards on screen (I had to read this 6 times to make sure it made sense, not sure if it even does at this point)

A screen shot of a computer program

Description automatically generated

### Changing physics with gravity

#### Context

Now we can finally have deceleration on the ceiling.

#### 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.

### 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. The exception would be that 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 occurs).

#### Implementation

The implementation wasn’t all that hard for the most part, consisting mostly of flags and testing for directions.

For the timing part, which I was previously somewhat scared about since adding pauses in the middle of a main loop tends to reduce the frame rate and smoothness of rendering, Jarrett gave a good suggestion (which apparently should have been assumed knowledge) to use a variable as a tick timer which is incremented every execution of the main loop, and would act as a flag for reverting gravity once its value is too large.

There was one problem that arose with resetting the direction of gravity after some time. Due to the direction of gravity reverting to its initial direction after some time, collision did not work properly.

And so, this resulted in this abomination of a nested log where I try to explain it in detail.

### 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.

#### Implementation

I was too lazy to think so I just added extra switch cases to detect collision in absolute directions after gravity was reverted. Not the prettiest or most efficient, but it is guaranteed to work.

And after that I’d assume that gravity has been now reworked properly.

### 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. This would be addressed [later](#_Restricting_Jump_+).

#### Implementation

I ended up scrapping the entirety of deceleration, and replacing it with a neutral system. Instead of applying deceleration on the object every tick, the object would only slow when no direction keys are being pressed. With this new approach to deceleration, the box can start moving as quickly as it would be able to stop moving.

I ended up creating functions to force all direction keys to be cancelled and to detect whether any direction keys are being held down, which would be probably helpful for the future.

### 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

So, I changed the activation key to space.

I guess the only interesting thing to mention here is how event.key for space is literally just “ “.

## 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 from year 11, 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.

I could’ve probably had some useful info prior to this, 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.

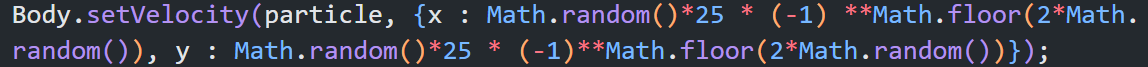
The problem with this method, after some testing, I realised was that the particles could only have positive velocity, i.e. they could only go rightwards and downwards. While I could fix this by writing up or even copying someone else’s separately written random function, I was too lazy and decided to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

### 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 in if statements. 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 0, it will not cap velocity.

That ran into another problem, which is that velocity immediately gets reset after jumping, due to 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 they would retain the uncapped velocity, awarding more precise inputs.

### 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

### Gravity Change

#### Context

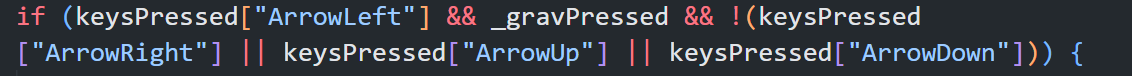
I am yet to implement diagonal gravity changes, and most likely won’t simply due to its chaotic nature.

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.

## Log 7: Camera

### How does the camera work?

#### Context

As the title states, this was basically me trying to wrap my head around how the camera in matter.js worked, so that I could use the camera for level transitions and screen shake effects.

#### Implementation

First of all, there was this lookAt function under render which would have the camera follow an object. While this probably wouldn’t be used in the final product, it would give me a good idea of how the camera worked, *right*?

A screenshot of a computer

Description automatically generated

Yep. The camera got resized to the size of the player block, so the player can have full awareness of the different shades of yellow used to colour in this 40 pixel 40 pixel square, just what we want to see.

So I passed a 400 by 400 pixel padding into the function and tried out what happened.

A screenshot of a video game

Description automatically generated

I wish I could insert a GIF here to show how bad it is

Slightly more promising. At least the map is somewhat visible? The stopping and starting was very rough though, and I felt like my head was getting slammed onto the actual wall every time the box collided with any object. After realising for the first time I suffered 3D motion sickness playing Portal, I have now discovered that I have 2D motion sickness playing whatever this is.

After messing around a bit more and reading the source code for matter.js demos, I found that the camera was defined by render.bounds.max and render.bounds.min, both of which containing a record of a coordinate respectively for the bottom-left and top -right of the camera’s visible range. I have no clue why this wasn’t documented anywhere else.

A lot of calculations would need to be run before actually modifying these bounds though, as the bounds do not preserve the original display ratio.

A screenshot of a video game

Description automatically generated

#### Sources

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

Used to figure out how to use a tracking camera.

* By [liabru](https://github.com/liabru)
* Accessed 22/05/2024

[Matter.js demo – views](https://brm.io/matter-js/demo/#views) ([source code](https://github.com/liabru/matter-js/blob/master/examples/views.js))

For understanding how to change bounds of the camera.

* By [liabru](https://github.com/liabru)
* Accessed 22/05/2024

### Screen Shake (Death)

#### Context

Screen shake effect to make the death animation more impactful.

The idea would be to just shift the camera around in some form of pattern upon death to emulate the ground shaking from collision (or something along the lines of that).

First of all was determining how the camera shake would be modelled. For the bare bones (linear model), the camera would move right for a set number of ticks at a constant speed, and then move left at the same speed after those ticks, rinse and repeat. After nearly throwing up from watching the linear camera follow however, the linear model was rejected.

Next up was the trigonometric model, which tossed values into the sine curve.

A screen shot of a graph

Description automatically generated

The huge advantage of this was that there would be no if statements checking when to move forwards and backwards, as the values would automatically fluctuate between 1 and -1. Additionally, this meant that as long as the duration was a multiple of the period of the sine curve, the camera would move back to its original position, avoiding the potential problem of the camera shifting slightly every time the player dies, and they just don’t get to see the game anymore after too many deaths (would be a fun idea though).

#### Implementation

So, I decided to try putting the sine curve in for the death animation. Sadly, I can’t insert a video into this so just imagine based on my description below.

While the trigonometric model would probably feel more fluid than the linear model (which I did not bother to implement), it didn’t quite feel like screen shake, instead it felt more like someone tying a camera onto the edge of a hamster wheel and let it spin while filming a set, with the video turning out to be the set just rotating across the screen, instead of a more rumble or vibration effect.

I figured that a big reason for this was the lack of dampening, considering that shaking would probably decrease in intensity over time in reality. So I went back the drawing board and graphed up more models to take dampening into account.

My first thought was to try the famous , which dampened exponentially over time.

A screenshot of a computer

Description automatically generated

The problem was that it was too drastic. One the left hand side you would have stupidly large values approaching infinity, and on the right hand side you would have infinitesimal values approaching zero. Translated into camera movement you would see the camera fly into outerspace, fall back through down through the centre of earth and come out the other side within a split second, and then it would move so slightly that you would barely notice its movement. Not ideal. For more of a middle ground, I then tried .

A screen shot of a graph

Description automatically generated

This had more moderate dampening, and the entre right half of the curve could be ignored with an if statement as the timer hits too great of a value.

With a bit more of shifting the curve around the model was put into code, and it looked amazing overall. The one tiny problem was that the curve was no longer symmetrical about the x-axis, and so the camera would not return to its original position, resulting in having to snap the camera back to its original position afterwards abruptly. That should be solved once the level transition is completed and I can just take that code and paste it here.

Also, after not having to deal with it for the project so far, pass by reference finally struck back by preventing me to store bound values which are records. Not too hard a fix thankfully. Only took me half an hour identify the cause of the error.

A black background with white text

Description automatically generated

### Level Transition

#### Context

This section below contained significant quantities of boring debugging due to errors including but not limited to forgetting to define variables, messing up scope of variables, having variables that would be undefined due to the nature of loops, forgetting pass by reference, forgetting double equal signs for conditions, forgetting that js can only do comparison with two values and missing semicolons that have existed since two months ago. For the sanity of both the reader and writer, the writer has decided to keep recounts of such errors to a minimum.

#### Implementation

Before a level transition were to be made, there would need to be a level. So, I quickly wrote up a level class including the bounds of the level so the camera knows where to be, a spawn point and an array of exit points to trigger level transitions, alongside methods to check if the player is out of bounds or at an exit point, with a global array containing all level objects because I already had too many globals to care.

Afterwards, was triggering the transition. When the checkLevelComplete method returns a coordinate when the player is within range of the exit, the current level would be set to the next, starting a transition timer and setting a flag true to indicate the transition taking place. As per the timer, the bounds would then be shifted by the total distance to be travelled divided by the total allocated ticks for the movement per tick (basically calculating average speed).

The above didn’t work however, even after fixing a lot of miscellaneous errors, (see context) and I asked for help from Jarrett one day. After scolding my codestyle for having new lines between if and else, having spaces on either end of the colon in records and having not enough functions, he took my laptop and started writing code.

The problem ended up being me defining a variable within the main loop, which resulted in it being overwritten every tick as undefined and breaking the program. As per classic js, there was not angry red words yelling that there was an error even though there clearly was, and the program still ran, somewhat.

He then proceeded to do some destructuring sorcery for the bounds to avoid pass by reference and put them into the tween function, and went on to teach me how to add an ease transition to tween between camera bounds.

We came into a few problems such as changing the distance to a proportion instead of just a number to make it fit the ease curve we copied and how to not have the distance between current bounds and target bounds not being rewritten every tick making it so that the transition would never actually reach (think cutting a rope in half forever – you’ll have something remaining always). After much brute force with neither of us really knowing exactly what was done, the code worked.

Oh by the way I started using a code formatter as per Jarrett’s suggestion.

#### Sources

##### [Easing Functions Cheat Sheet - easeInOutSine](https://easings.net/" \l "easeInOutSine)

Copied entire function.

* By [ai](https://github.com/ai)
* GPL-3.0 license. Permits all distribution and changes of the software for both public and private use. License conditions were not violated by copy pasting the code.
* Accessed 1/06/2024

### Screen Shake (Gravity Change)

#### Context

Self explanatory.

I seem to be getting to the point in the logbook where all the context seems to be already explained in previous parts.

#### Implementation

This one was an annoying one. Not hard, just annoying.

The screen shake part was as basic as it could be. In fact, I literally copied the death screen shake as a placeholder for the time being. It was the things afterwards that was annoying.

For starters, dashing across screen transitions would cause the camera to just give up. The game would still be running at full speed no delay, just that the camera would refuse to move. The source of the error was obvious – the bounds of the camera being changed by both the level transition and the screen shake, which was never an issue before since it was impossible to die during screen transitions (I was very wrong about this. I will proceed to shoot myself in the foot about two logs away from here).

The solution was simple, add an if statement that refuses to do gravity screen shake during transitions.

Then a bigger problem appeared, the tween after the gravity change didn’t work, as if the function was never called.It was definitely not a problem with the tween function per se, since it would work perfectly fine for level transitions as well as after death screen shakes.

Since I had little to no idea how the tween worked in the first place, the debugging was mostly me commenting out lines and seeing what would happen if this specific line didn’t exist. That didn’t get me very far so I went back and logged every single variable that was involved. I could do a desk check at this point but it still probably won’t be worth it, especially as I have no idea how to read the dark arts of destructuring (thank you Jarrett) (also Microsoft Word doesn’t think destructuring is an actual word).

After way too much logging, the problem was inconsistent use of variables for bounds. While the library had a property constantly tracking the bounds, Jarrett used destructuring which avoided pass by reference, and made a completely object from the library’s property, which was the basis for the entirety of tweening (I think at least).

After messing around with defining the variables a bit more, I put in a function to update these variables after the tween function would be called in case I forget in the future. This tiny bit of foresight I have has saved me so many times in this project, I am incredibly grateful at this moment that the movement I wrote months ago still works without me having to interfere with code I completely forgot about.

As I spent more and more hours debugging a stupid screen shake animation, my sanity had slowly left my soul, resulting in some terrible life choices, such as actually paying attention to the animation and realising that it didn’t produce the “swoosh” effect I wanted but was instead more of a “rumble” (yes this is how I will be referring to these from now on).

Out of complete OCD, I booted up Celeste and spent 5 minutes staring into the screen to figure out how they handled screen shake for their dashes. My findings were:

* The screen shake effect is different for horizontal and vertical dashes
* The screen moves by a visible amount in one direction and snaps back fairly quickly
* Certain elements have a glow-ripple sort of effect upon dash (I don’t think I’ll be able to implement anything like that)
* Annoyingly, it seems the screen shake effect is the same for both dashing left and right, the screen moves rightwards initially regardless of dash direction. Similarly for up and down. I can no longer sleep in peace.

This made a lot of sense, since the death animation was intended to be impactful and somewhat “rumbly” to rub in the fact that the player died.

So all I had to do was add in a switch to move the screen differently for different gravity directions. The screen shake wouldn’t even need to be that fancy, linear would work completely fine, *right?*

No. Of course not. Never would it be that easy. Despite the code clearly saying it was a linear shift, the camera moved nowhere close to smoothly, making a stop halfway through and then moving again. Equivalent to a “rumble” even though the code said it was a “swoosh”.

So, what happened here? After searching on the internet with no result and people saying that it would be better if I used a separate library for the camera, I went back to changing variable values and seeing what would happen.

Guess what? The code actually does work. You see, if you just turn the duration of the screen shake up ever so slightly by some 10 ticks, the animation is perfectly smooth, but if it just goes slightly below that turning point IT BECOMES CHOPPY AGAIN.

I was forced to conclude that this is just a library problem that has no way to be fixed after hours of messing with values. At the end of the day, the screen shake doesn’t even look that different to how it was originally…

### Transitioner

#### Context

Previously, the screen transition would be detected using hard-coded values around a given point. This could be quite inconsistent and annoying to implement in mass, and has at times been inconsistent with its triggering.

The bigger problem would be transitioning back to the previous room. Using hard-coded coordinates, it is likely that the camera would transition back to the previous room immediately upon entering the next room, and vice versa, creating an infinite flashing between two rooms without enough time for the transition to actually take place.

Using body objects made with matter.js would somewhat solve this problem since the dimensions of collision would be easier to change, and these transitioner objects will only be functional upon fully entering the new room (i.e. they would be removed if not in the room), preventing the flashing loop. I could also use a timer or another if statement to check if the player has left the spawn area so that it would not take the player back to the previous level immediately, but that just seems to be too much effort.

#### Implementation

First and foremost, the problem that would have to be tackled is collision. In matter.js normally, collision with an object forces movement to halt, and it is impossible to pass through an object. There should be a way to change that, hopefully…

Good news: there is. Bad news: It was hidden deep within the documentation.

Hidden away in the corner was this demo called “sensors”, which had a ball falling through a block, changing colour if it had collided within the block.

A screenshot of a computer

Description automatically generated

Despite so, I still had no idea what disabled normal collisions, so I copied the entirety of the code, tweaked it a bit so that it could link a html file and run locally, and started a bunch of console logging and commenting out lines.

Eventually, I came to the conclusion that the isSensor property for a body was what determined collisions. If it is set to true, the body will pass through any other bodies, and the collision event will still be triggered despite it not losing velocity due to it. In which case, it would be set on the transitioner rather than the player, so that the player can pass through only these transitioners and not every single obstacle in the game.

So I quickly wrote up a transitioner class and put them in the game.

A screenshot of a computer

Description automatically generated

I tried to make them be removed from the world upon entering a new level, but for some reason that didn’t seem to work. I then went back to the demo file and messed with that for a while. I got nowhere, but, I got this really cool error message by forgetting to write a comma:

A screenshot of a computer

Description automatically generated

It wasn’t too big a problem though. Even though the transitioner still exists, I can still just treat it as if it didn’t exist, they are going to be invisible at the end of the day anyhow. For the infinite loop situation, I could just make the transitioners thinner and space them well so that it’s impossible to trigger both at the same time.

And then another problem occurred. The box managed to die during a screen transition. Due to never accounting for this, the camera just absolutely flew off the screen into oblivion. Thankfully, the tween transition still works properly and the camera is able to teleport back to witness the death.

A screenshot of a computer

Description automatically generated

Due to the trasitioners being too thin, the box didn’t actually trigger transition across the screens, and was considered out of bounds, dying immediately.

There was no great solution to immediately fix it, so it was a lot of messing with dimensions to make sure it would have to cross the transitioner. After a bit of testing, it never happened again so I hope I fixed it.

Another thing that I only realised happened after creating the new level after transition was that the initial gravity change screen shake after a screen transition would not work properly going off screen. This would specifically only happen if the gravity change occurred before touching ground after the screen transition, and the screen shake would be perfectly fine once the ground was touched.

It turned out to be a problem with the flag for setting bounds never being actually reset due to the tween function running additional checks besides the necessary ones already implemented in the main loop, making the bounds not reset properly during the screen transition. After removing the additional if statement, it worked perfectly fine.

#### Sources

[Matter.js demo – sensors](https://brm.io/matter-js/demo/#sensors) ([source code](https://github.com/liabru/matter-js/blob/master/examples/sensors.js))

For understanding how to use sensor bodies. Downloaded and modified for learning purposes.

* By [liabru](https://github.com/liabru)
* Accessed 07/06/2024

## Log 8: Falling

### Falling Platforms

#### Context

The biggest problem I have with the project right now is the fact that the concept of “gravity change” isn’t entirely emphasised with the gameplay. There isn’t much really to indicate the a gravity change has happened rather than just a normal dash.

The intended solution for this was adding obstacles that would move according to the direction of gravity.

#### Implementation

I thought falling platforms could just be a subclass of the normal platforms, but a few problems would happen if that was the case.

Most of this arises from the fact that I have to detect the direction of different walls, which resulted in a singular platform actually be formed by a up, down, left, and right component, each detecting collision for one direction.

As a result, the platforms are essentially composed of four sticks that collapse immediately with the force of gravity.

A screenshot of a computer

Description automatically generated

In case you were wondering, making the sticks thicker does not solve the problem either, it still collapses.

A screenshot of a video game

Description automatically generated

I’ve also messed with things such as setting inertia to infinity which prevents rotation, increasing friction values to be stupidly high, but ultimately the physics engine would not prevent it from collapsing.

Alternative solutions would include using constraints (basically rubber bands) to hold the sticks together, but from what I’ve read on their documentation, they still act as joints for the sticks, so unless I make the platforms triangles (never), they will still tend to collapse.

So that idea goes out of the window now. Instead the falling platform will just be a block by itself.

At first the falling platform would count as a floor, ceiling, left wall and right wall simultaneously. This caused some very weird situations such as jumping when colliding with the side of the platform, which allows reaching incredible heights since the game thinks the player remains on the ground for a while.

I gave up on that idea and went to make the falling platform count as none of the above. This was mostly sacrificing functionality for stability, as the player would lose the ability to jump and reset gravity change on the falling platforms.

A screenshot of a video game

Description automatically generated

### Falling Spikes

#### Context

Same idea as falling platforms.

#### Implementation

I thought to do this as a subclass at first as well, but again a few problems made that not possible.

Despite the stationary spikes seeming like they are connected to each other, they really aren’t. The moment you let them fall they split up and fly all over the place.

A screenshot of a video game

Description automatically generated

And realistically, there wouldn’t be a line of spikes that are falling, considering it would be especially weird to have a line of vertical spikes standing on the ground after gravity change. As a result the looping to create rows of spikes was completely unnecessary for the falling spikes.

At this point it was simpler to just rewrite the entire class for falling platforms specifically, so that’s what I did.

A screenshot of a computer

Description automatically generated

### Resetting & Sleeping

#### Context

There were two big problems associated with moving between rooms and falling objects:

The first was resetting. After leaving the room and coming back or dying and respawning, the falling objects should return to their initial position, serving as a retry mechanism.

The second was stopping movement. Oftentimes I would have objects of the previous room falling into the next room.

#### Implementation

For resetting, the process was literally creating a new method in the class to set the position of the falling objects back to their initial coordinates.

For stopping movement, matter.js conveniently had a Sleeping module which could pause the movement of certain objects. So, all that then had to be done is assign each falling object to a room and detect whether or not after transition the player will still be in that room.

I initially tried to use records to with the room number as the key and falling objects as values to store them, but realised that arrays would achieve the same purpose if the key was just an integer, so it instead turned into an array of arrays.

A screenshot of a computer program

Description automatically generated

#### Sources

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

Used for reference in learning about the Sleeping module.

* By [liabru](https://github.com/liabru)
* Accessed 26/06/2024

## Log 9: Menu

### Menu overlay

#### Context

The first most noticeable feature of a pause menu would be that usually the game is dimmed, and big text saying “PAUSED” would appear on screen.

I also needed a menu so I could get some more marks on customisable GUI.

#### Implementation

The idea would be to have an absolutely positioned div layered on top of the main canvas, with some form of translucent colour.

I spent about half an hour trying to figure out why my div was invisible, until I realised I was changing its color style rather than its background-color, making only the text change colour and not the actual div…

Also for some reason the colour of text is inherited from the div and cannot be set directly by the button, so I put the colour of the text inside of a span so I could change its colour on hover.

A screenshot of a computer

Description automatically generated

### Pausing

#### Context

After a menu shows up with options, the game should probably actually pause itself, instead of having things still moving in the background.

#### Implementation

The most intuitive way to do this would be to set timeScale to 0 just as it was during screen transitions, freezing all objects in place, which I did.

The problem however is that the main loop will be running once per frame regardless of the time scale. This problem was most noticeable during the transition and death animations. As both of these involved manipulating the time scale in some way themselves, their time scales would override that of the pause functions, resulting in objects still moving in the background despite being paused.

I tried using cancelAnimationFrame to pause the execution of the main loop, but for some reason, probably due to the library, this would result in objects not being rendered and the block disappearing from the screen, not returning even when the main loop is called once more.

Instead, I ended up just making it so that the pause function could not be executed during these “cutscene”.

Besides the cutscene problem, the main loop not actually being cancelled resulted in keyboard inputs still being registered during pause. This was not the most significant problem, since the box would not be able to move anyhow since the time scale is 0. The arguably more problematic situation was that gravity change could be triggered by pressing space on the pause menu.

After a while of thinking, I figured that there was no good solution to this. Since keyboard inputs would not be registered unless on the game screen, players are still free to tab out and use other windows as required, not interfering with the game. So, this was then left as a feature.

### Wireframes

#### Context

This was the easiest thing I could think of to make a customisable GUI.

#### Implementation

There is a built-in option into the renderer of matter.js that enables wireframes. At first, I thought this would require re-rendering everything on screen and probably some form of soft-refreshing of the webpage, but it turns out it could just happen.

However, when turning on wireframes, the previously invisible transitioners would now be visible. This was since I had set the fill style of the transitioners to be transparent in non-wireframes, which would be ignored when rendering in wireframes.

A screenshot of a computer

Description automatically generated

Thankfully, there was this nice visible property which would do the same thing regardless of wireframe or non-wireframe rendering styles and solved the problem.

A screenshot of a computer

Description automatically generated

#### Sources

##### [Matter.js Documentation – options.wireframes](https://brm.io/matter-js/docs/classes/Render.html" \l "property_options.wireframes)

Used as reference for wireframe rendering.

* By [liabru](https://github.com/liabru)
* Accessed 27/06/2024

##### [Matter.js Documentation – render.visible](https://brm.io/matter-js/docs/classes/Body.html" \l "property_render.visible)

Used as reference for wireframe rendering.

* By [liabru](https://github.com/liabru)
* Accessed 27/06/2024

### Pause when focus lost

#### Context

Due to how matter.js works, every time the browser window is out of focus and requestAnimationFrame is no longer calling the main loop, all non-static rendered objects disappear.

The work around for this would be to pause the game when the window loses focus.

#### Implementation

JS had this handy method on document called hasFocus, which returns whether or not the window is focused. Put that in an if statement and link it up to the already made pause function and it is done.

Not entirely it turns out. The special cutscene cases for pausing come back to mess with this. If the player were to tab out during the death animation of screen transition, the game would not be paused at all.

To account for this, I added another flag (I really have too many global flags at this point) to determine if a pause input has been received. If due to cutscenes the pause cannot be activated, the flag will remain true and using an if statement on this flag the pause function would be called after the cutscene has ended, similar to how buffering a keystroke works in games.

Another tiny change was that I added a game start menu, which was literally the pause screen but with the “resume” button replaced by “begin game”.

#### Sources

##### [Stack Overflow – Check if window has focus](https://stackoverflow.com/questions/17389280/check-if-window-has-focus)

Used for reference on the hasFocus method

* Answered by [James](https://stackoverflow.com/users/82586/james)
* Accessed 28/06/2024

## Log 10: Final Level

### Pretty much all of the final level

#### Context

Having two levels doesn’t feel quite enough to flesh out the concept of the game, and ideally I would have one new mechanic being introduced to the player per level. So, a third level was something that I decided to do.

The first level would be basic movement as well as platforms and spikes. The second level would be falling spikes, and the last would be falling platforms.

#### Implementation

After a few hours of setting the location and dimensions of objects, the end product looked like this:

A screenshot of a video game

Description automatically generatedA screenshot of a computer

Description automatically generated

The idea for the level was that the player would have to switch gravity around so that the falling platform would be able to cover the spikes. After tweaking the distances slightly, the level could also be completed with a precise wavedash.

As seen in the wireframe version of the level, it consists of many overlapping objects to ensure the essential parts function as intended. The property-based collision system that was implemented ages ago resulted in some problems. For example, the player would be able to jump when collided with ground. However, this does not necessarily have to be the player being *on* the ground, they could also be *below* or *besides* the ground to jump. While this problem does not show with simpler level designs, such as the previous two levels, it becomes noticeable in this level.

The solution to this would be to apply collision filters depending on the direction which the box is moving. It probably will take considerable time to implement and debug as any other part of movement did, and so I think I’ll leave it as an intended feature for now.

Lastly, I added another transitioner at the end of the last level which brings up the pause menu, replacing “pause” with “demo complete” and bringing the player back to the first level.

### More on the final level

#### Context

I take back everything I said about my poor collision detection. I just realised that I made the platform class have four unique sides for a reason, all I have to do right now is replace my walls with platforms and it should work perfectly fine.

#### Implementation

Replacing some of the more problematic walls and floors with platforms, the level now looked like this:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

After having wall jumps and gravity changes actually working properly on this level, I then realised that it would be trivial since the whole level could be skipped by simply changing the gravity to be leftwards permanently. I then made (a lot of) adjustments to account for this:

A screenshot of a video game

Description automatically generated

A screenshot of a video game

Description automatically generated

The option of changing gravity leftwards still works here, but would be a lot more difficult considering the player would have to stop midway and then change gravity upwards without dying to spikes.

There was also a tiny problem with the tween from this level to the first after completion that was caused by camera movement of the player dying overlapping with the screen transition. So, another flag, this time for controlling if the player would die from being out of bounds was added to prevent this.

### Title

#### Context

#### Implementation

#### 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

○ 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