# Munchkin Rescue – Report

Jordan Harrison

Contents

[Game Summary 3](#_Toc26963108)

[Storyboard 4](#_Toc26963109)

[Implementation Specification 6](#_Toc26963110)

[Research 8](#_Toc26963111)

[Critical Review 9](#_Toc26963112)

[Assets 10](#_Toc26963113)

[References 12](#_Toc26963114)

Repository: <https://github.com/JayAitch/CI328---Phaser/tree/master/TowerBuilder>

Live Game: <https://jayaitch.github.io/CI328---Phaser/TowerBuilder/>

Note: Pressing the C key will trigger level completion, this is to only to simplify testing and is not intended as part of the game

## Game Summary

Munchkin rescue is a casual browser based cross-platform game written in Phaser \*\*\*. The munchkin moves under the force of gravity and it is the players job to create a course it can traverse to get it to the goal in each level. The player does this by placing different sized blocks to build structure. This game is inspired by lemmings, angry birds and marble runs

The player only has a limited number of blocks they can place with up to 16 different block types which vary in size and shape. The level the player is on dictates which blocks and how many of them are available.

As the player progresses through levels the difficulty increases with more complex block placement required.

Difficulty settings add optional challenge to the game by reducing the number of blocks available to the player. This means they need to be more careful with how they place blocks and may also need to use a less appropriate shaped block to achieve the purpose of one that was, in lower difficulties, an exact fit. For example, see figure 1 and 2 for the different solutions required in easy and hard on level 2 both will result in level completion but on hard mode block placement requires more finesse.

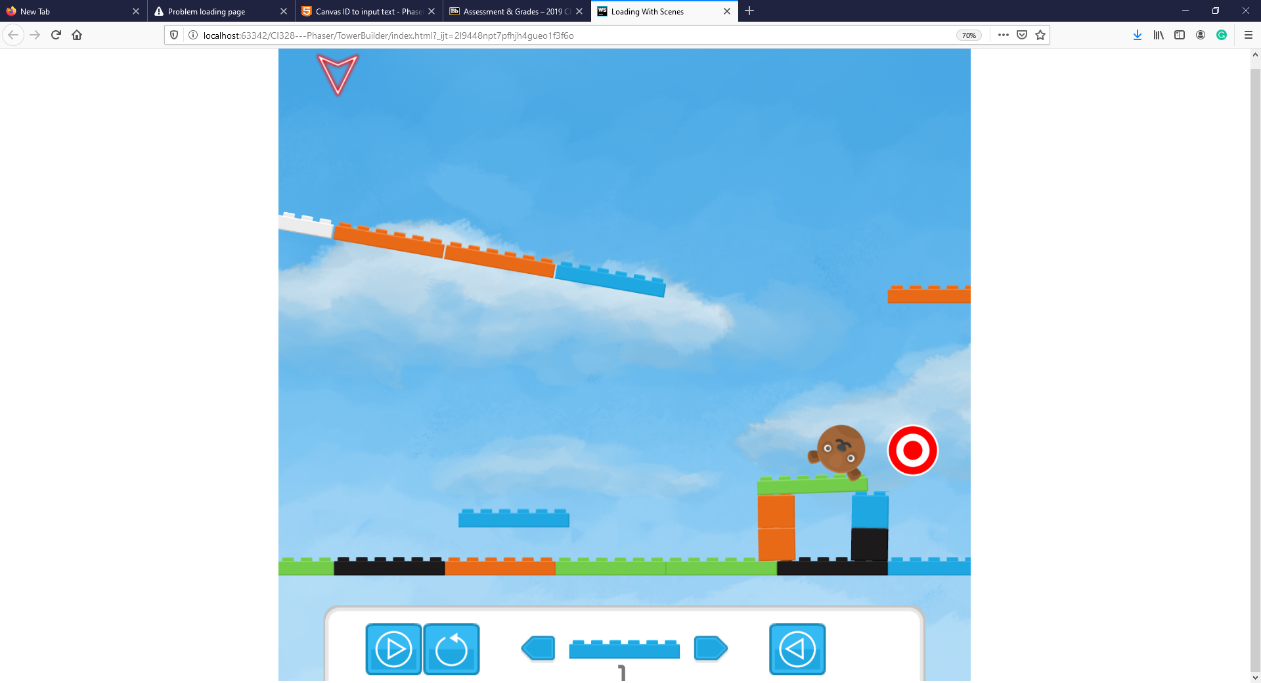


Figure 1: level 2 solution on easy difficulty

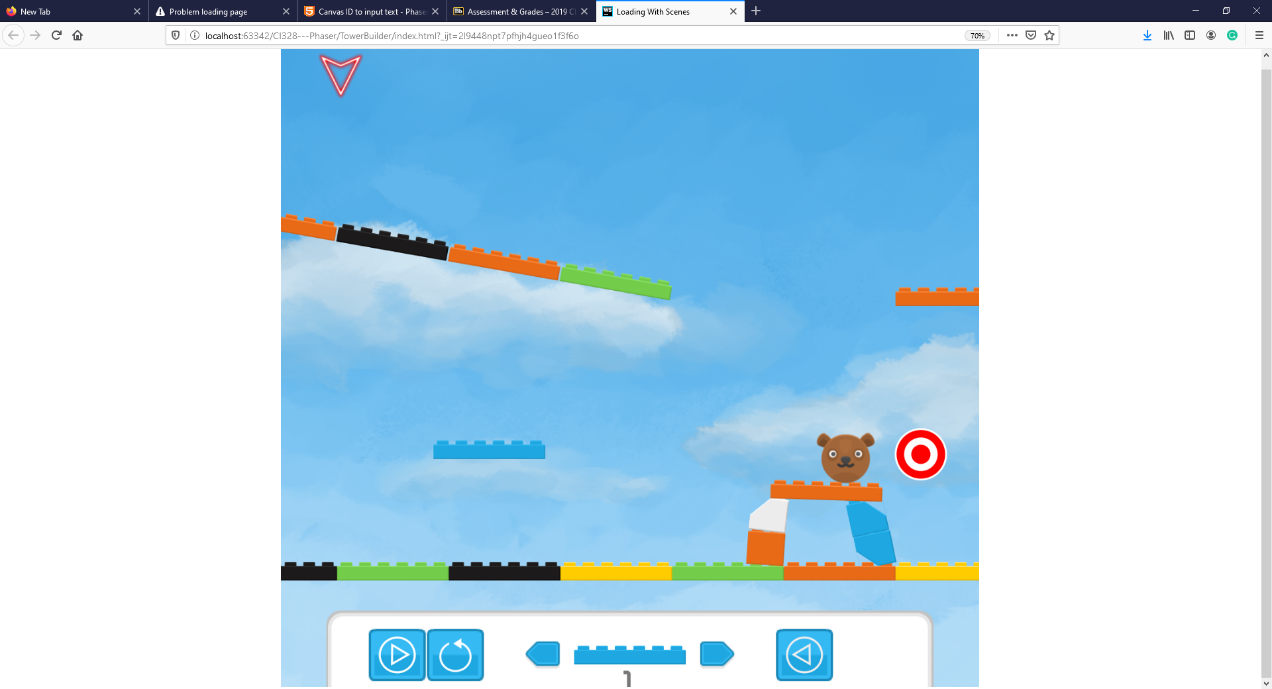
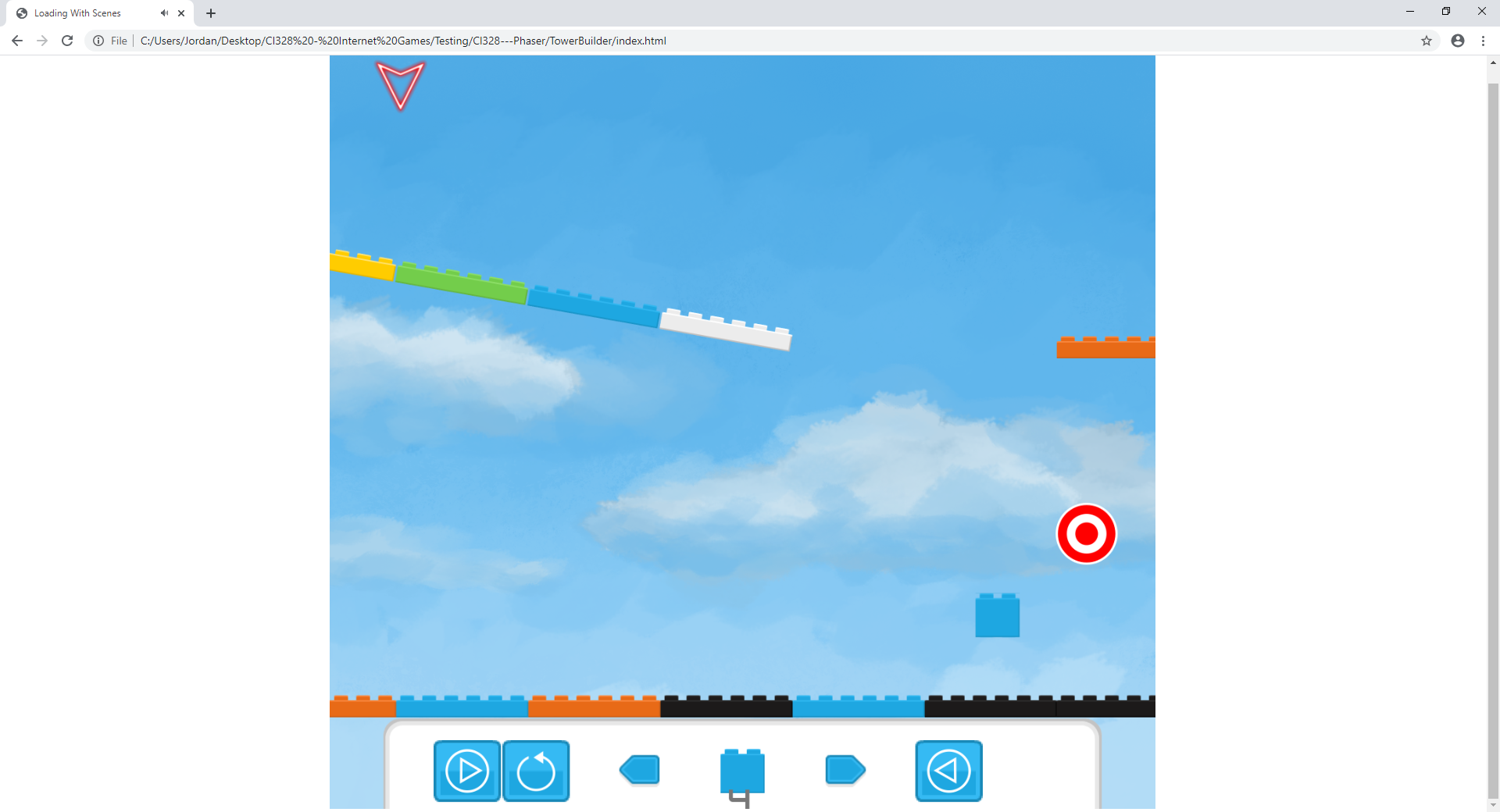
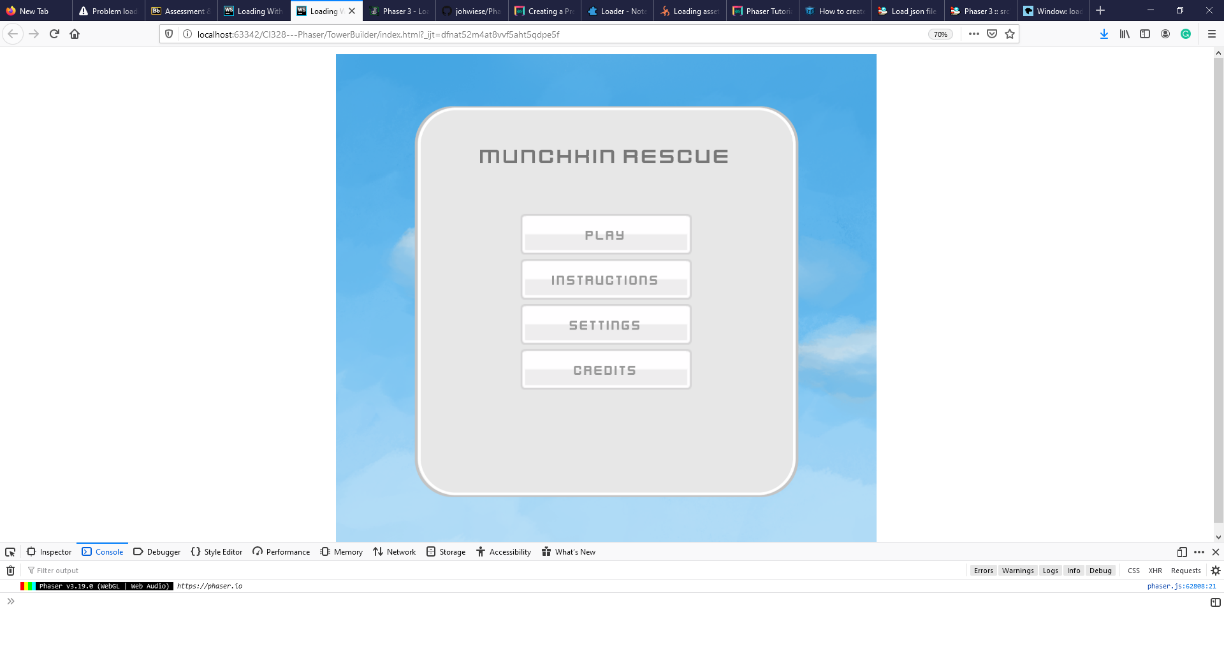


Figure 2: level 2 solution on hard difficulty

Reward is added through the addition of points, players will attempt to complete levels with as many points as possible. Score is initially given for completing the level, the harder the difficulty the higher the score. Resetting object placements or using munchkins will reduce the score. Having blocks left over will increase it.

# Storyboard



1 - Game Finished loading and displays landing screen

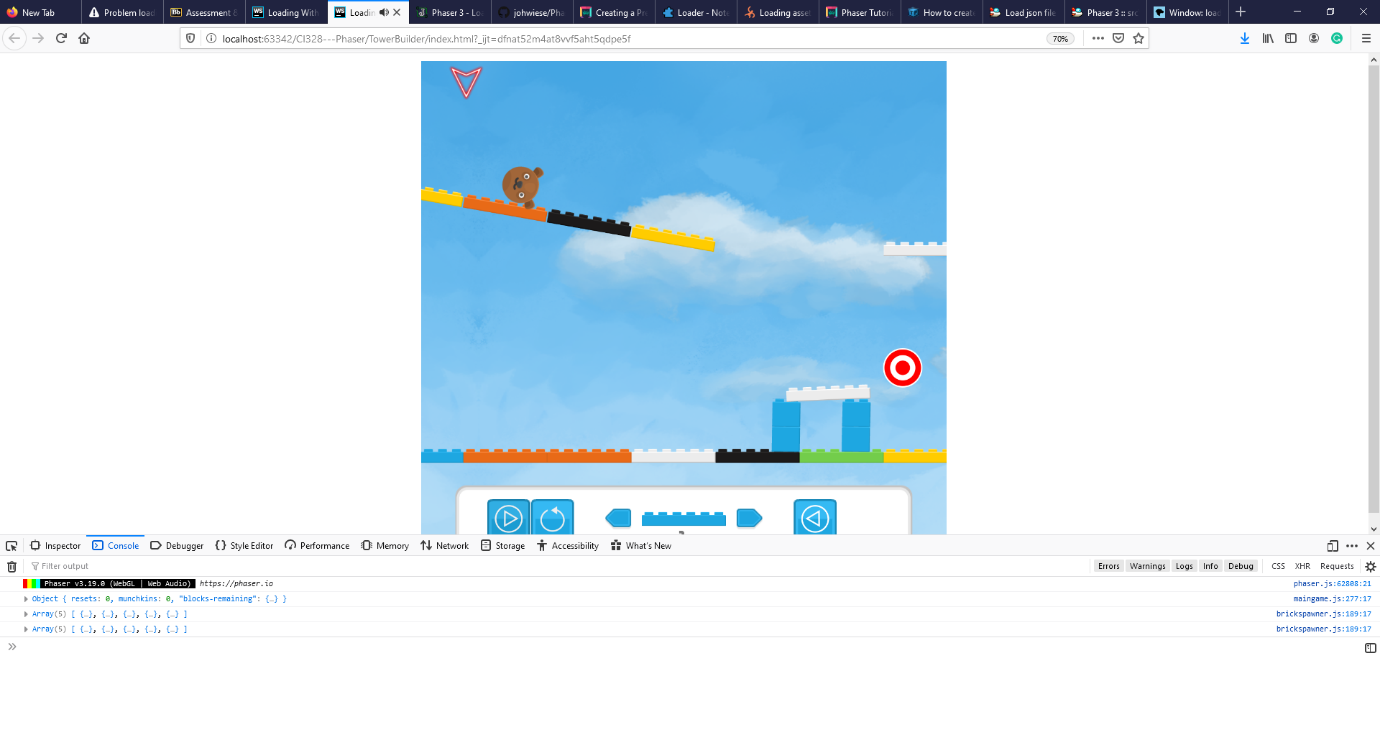
2 - Level is loaded by placing objects and making blocks available to the player. The blocks available are based on difficulty and level.

Current Block Cursor

Block display/ UI controls

Target/ Goal

Munchkin Start

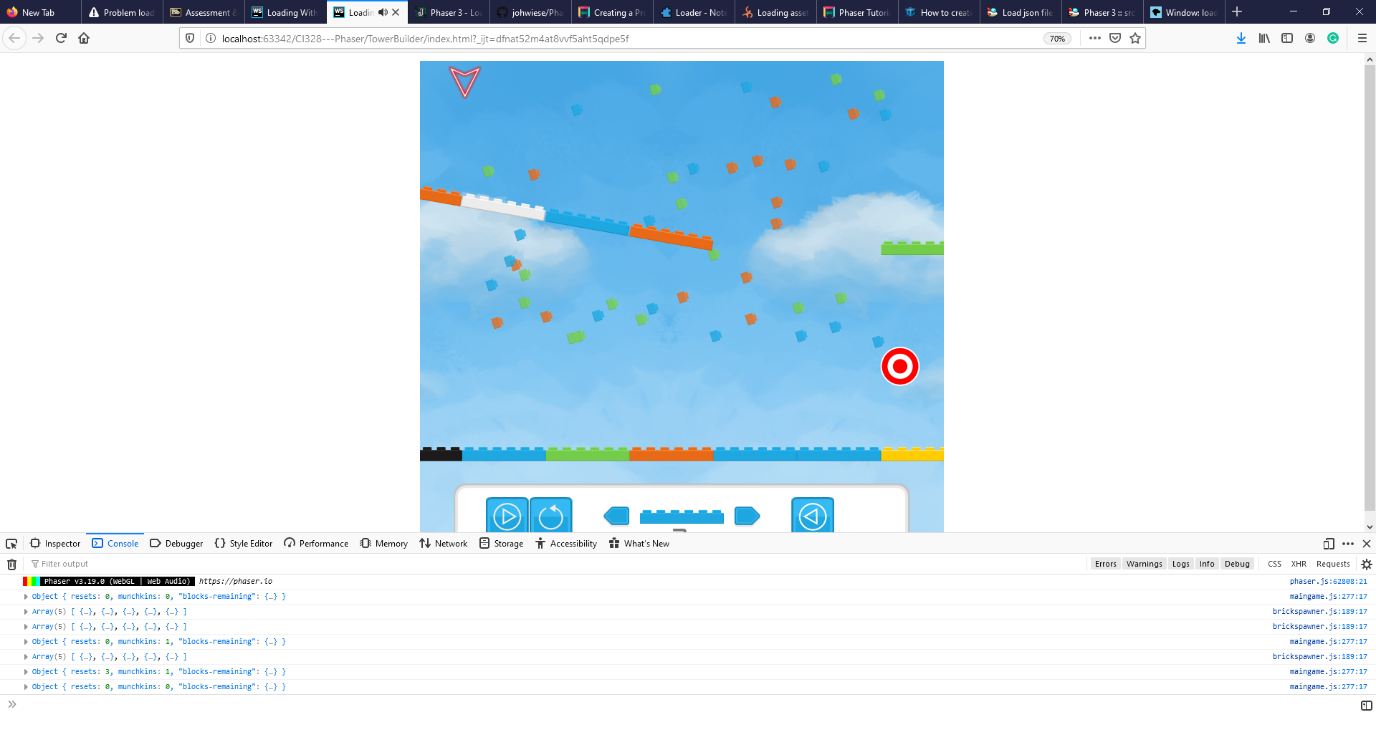
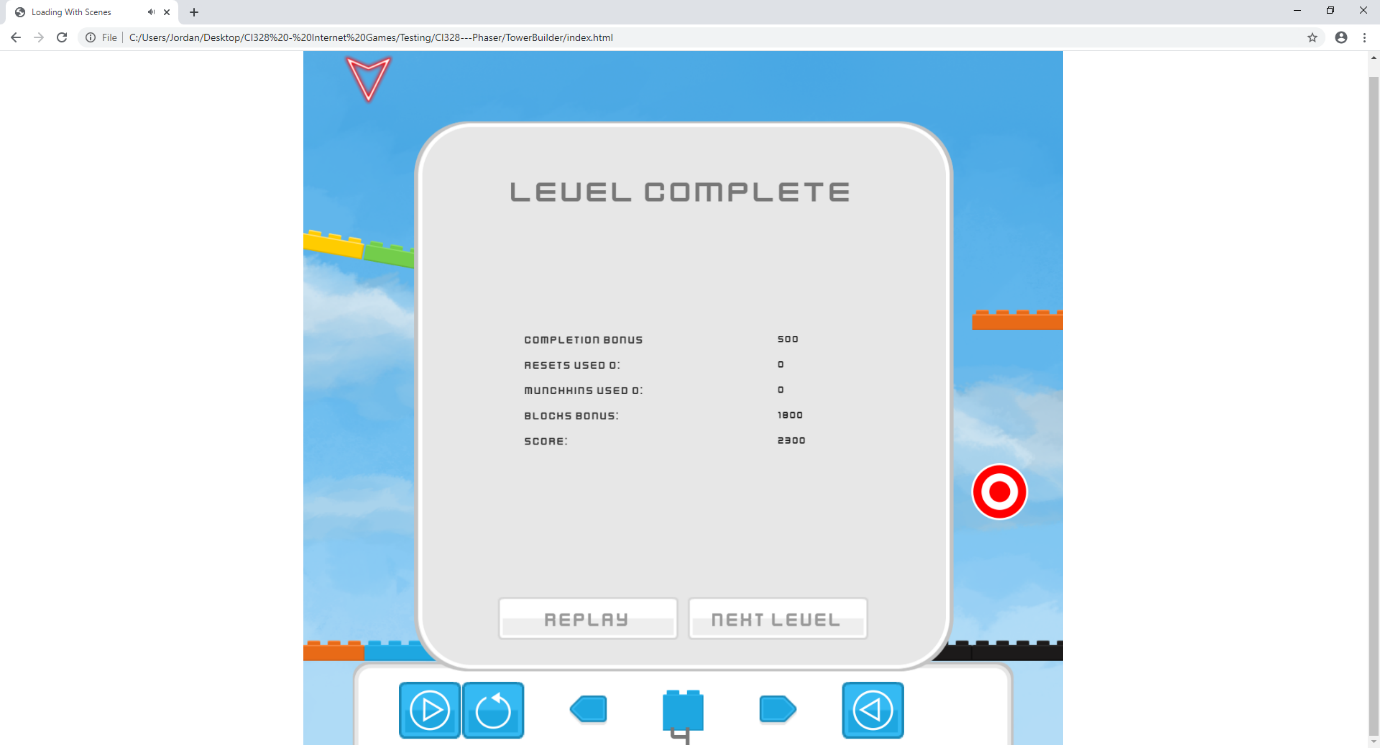


5 - Player releases a munchkin at first it travels the levels initial blocks.

4 - Player builds a structure they think will get the munchkin to the goal.

3 – Player starts placing blocks to help the munchkin travel to the target





7 – Munchkin reaches the goal and the Level completes playing fanfare music and showers block particles

6 - Munchkin successfully traverses the level, over the structure the player created, to the target.

8 - Level complete screen triggers showing the player the points they have earned.

# Implementation Specification

This first scene that is created is a boot scene. This caches any assets required by the loading screen including configuration objects for any assets loaded thereafter. As there is no display on this scene there needs to be minimal operational cost so that it finishes loading and switches scenes before a user perceives it.

The second scene that is create is the preload scene, this allows all assets to be preloaded from the server filesystem and stored in the browser cache. Asset file locations and runtime key references are defined by a JSON. This separates data from code and allows for new assets to be quickly defined. This design was inspired by an article written by Wiese (wiese, 2017), modifications have been made to interpret brick body definitions.

The physics world class contains both the munchkin and brick spawners. This allows additional actions such as UI updates, and score keeping, to be performed when new bricks/munchkins are spawned or the level is reset.

Any action triggered by a player is executed through the controller class, the main purpose of this is to create a single point of action for HUD button-based controls and key presses. This enhances continuity and reduces repeated code.

Level loading is inspired by the way Tiled stores map data. A JSON is used to represent the initial state of each level for all difficulties. The level object can then be parsed adding static and dynamic blocks to form the base level layout and assign the blocks available to the player. This allows game levels to be defined externally to code and simplifies the creation of a level creator.

The level complete and menu screens are implementations of phaser scenes. Control of loading and display is done through the scene manager API. This keeps a separation of game states; variables and UI objects are only required when the scene is being shown.

The game HUD class contains all functionality required by the HUD. This controls the position and image of the block placement cursor and any controls that are required for touch screen buttons to work. Any actions performed by HUD buttons are done through the controller object to maintain consistency in execution.

A major requirement of the design was for physics-based rotations to be enabled. A physics engine called Matter has been ported to phaser and is used for this kind of behaviour. This provides a system that acts on objects allowing the definition of complex bodies to enable some of the core gameplay mechanics.

As a knock-on effect of using the matter physics engine over phaser’s arcade physics colliders do not trigger call-back events. This somewhat complicates overlap checks as it must be done manually through looping over collision pairs. This is a more operationally expensive process but does not negatively affect gameplay.

When placing blocks it is important that there rigid-body does not intersect others, any considerable overlap causes the blocks to be effective stuck together and a large impact is required to break them apart. To check block placement the cursor object is used for positional and dimensional information. This is used to create an area to check for any other physical blocks before attempting to place. Previously this was done by calculating the position using the blocks physical definition but this was far more error prone.

Blocks rigid body definitions where created using Physics Editor inspired by Löw (Löw). This simplified defining block’s bodies significantly reducing errors and removing data structures out of the code making it more readable.

# Research

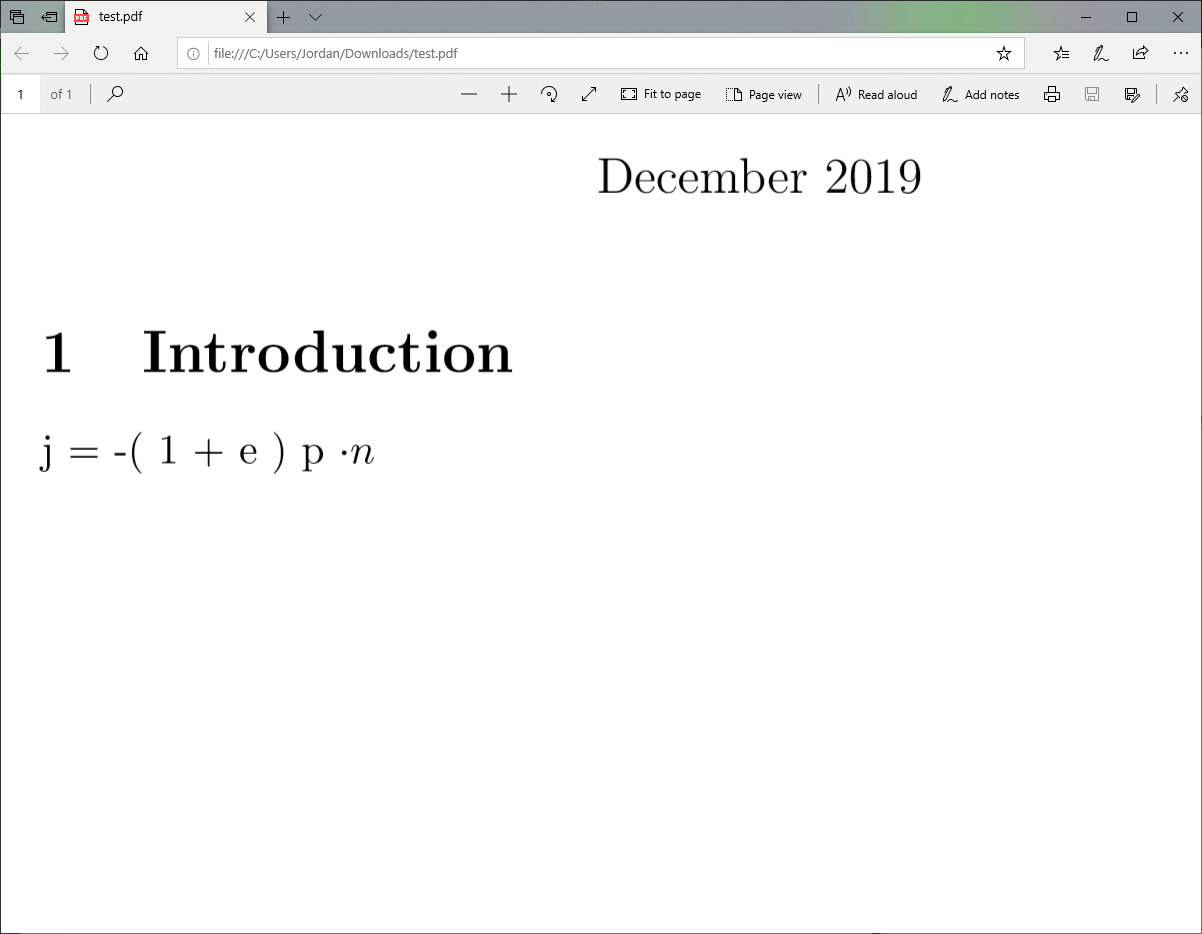
One of the major requirements of the design for the game is for the ability for objects to have simulated physics. Without angular velocity building would be significantly easier. The implemented physics simulation needs both angular and linear velocity that respond to retarding forces of air resistance and friction for the tower build mechanic have enough complexity to feel believable. Something which is a goal of game based physics engines (Jakobsen, 2008).

Phaser come equipped with 3 different physics engines: arcade, impact and matter, only impact and matter can simulate physics with enough complexity to create realistic movements out of the box.

Arcade in its base configuration only supports rectangle and circular shaped colliders. It has clean implementation of collision call-backs to allow control of gameplay relevant events when collision occur. This engine could be built on to simulate physics creating a realistic impression of movement. But collisions between complex shapes require implementation of some form of SAT (Seperating Axis Therem) based logic for example SAT.js (Riecken, 2019) to keep collision calculations accurate. This became too much of a development overhead despite the increase in performance Arcade physics would offer.

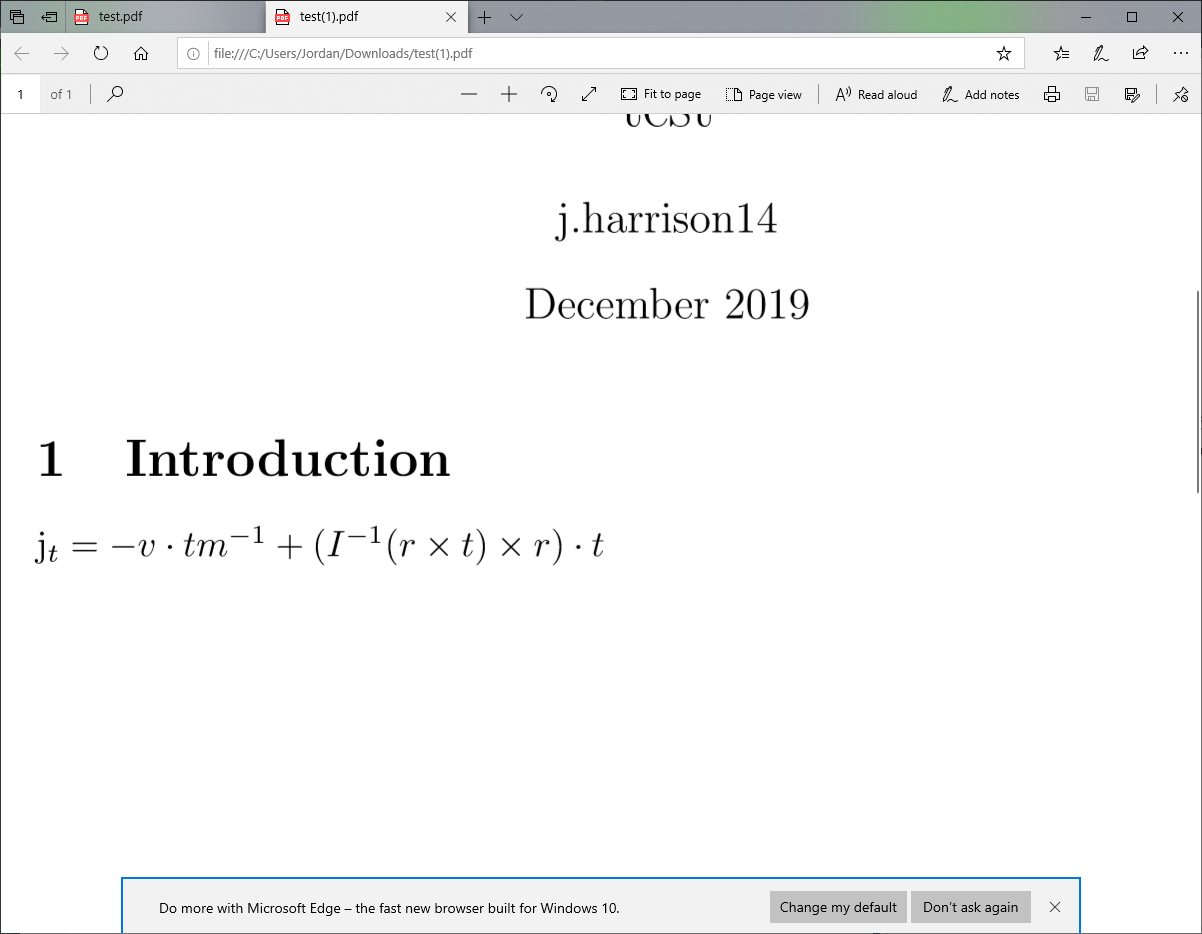
MatterJS is a physics engine written in JavaScript. This engine is a rigid body-based physics engine that uses a combination of Adaptive grid broad-phase detection, AABB mid-phase detection and SAT narrow-phase detections to resolve collisions (Liabru, 2016). This allows complex bodies to rotate whilst maintaining collision accuracy.

One method that can be used to calculate a collisions reaction is the Linear Collision Response (Jakobsen, 2008) this takes the restitution (e), the dot product of the linear momentum (p) and contact normal (n) of a collision to calculate the magnitude of the collision impulse (j).



This gives a magnitude impulse to apply to a body to create an effective reaction of a collision.

Believable physical reactions of collisions are understandably complex. A mechanism used by Matter to simulate linear friction is the Coloumb Friction Model (Jakobsen, 2008). This model calculates friction impulse using: friction impulse magnitude limited to a friction cone (jt), friction coefficient (u), the tangent of the direction of slide (t), the velocity at contact point of the body (b), the contact point minus the centre of mass (r), inertia tensor of moving body (I) and the mass of the body (m).



The result of this is the magnitude of a linear impulse that will slow the body down. This calculation cannot be used to calculate retarding forces required to reduce angular velocity.

Understanding how these algorithms work was advantages when making specific tweaks to objects physical properties. It requires a specific amount of maintained momentum to allow the munchkin to reach targets, especially on levels without rotated blocks.

# Critical Review

The interaction of json defined levels and the Level loading component is quite effective. With a simple understandable data structure levels can be changed without any understanding of the code behind the scenes. The separation into a septate json object contained on a different file simplifies future creation of a level editor. This design has saved time when making modifications to levels. The blocks a player has access to on each level are defined here enabling level more adjustment to levels than simply the blocks already in it and allowing different difficulties to be defined.

In order to make it easier for the players to stack blocks physics bodies where created with holes and pegs, like Legotm bricks. This enabled the gameplay mechanic of building structures from the blocks. Having played around with the mechanic with different size blocks and gravity it became a pivotal design feature that the rest of the game hinges on. It creates an obvious feedback to the player of success, by their structure remaining standing. The ability to remove the blocks and undo placing the last one enables the player to, attempt something, see that they have gone wrong, make a correction and learn from previous attempts to successfully make a structure. The game encourages a player to build structures that can support a munchkin getting to the level goal.

When completing a level there a sound and particle effect are used to create a fan-faire. This is creating a fun moment for the player, particularly in harder levels where it will take multiple tries to complete a level.

A significant issue with multi-platform use is the size of the canvas that is rendering the game. This doesn’t respect different form factors making the game harder to play on a tall narrow screen (like a phone). This issue can be circumvented by scaling the canvas to the browser window width respecting the devices pixel ratio, this will require a scaling factor to be applied assets inside the game. This will fundamentally change the way positioning works as locations of level objects will change depending on the size the canvas has scaled to.

Level design is restricted to a single objective completion, through hitting a target, although multiple targets can be added to a level. The game would have a significantly increased variety if there was a system in place to create levels that require the player to hit multiple targets in order to complete a level. In order to implement this a check would need to be performed when target-munchkin overlap occurs, before disabling that particular objective hitbox. This check simply needs to compare how many objectives are complete with how many that are needed to finish the level. A visual queue of the goals met could be shown to the player through the UI or by changing the appearance of an objective.

Currently levels progress from the first until the last. Once the player has progress to a successive level there is no way for them to return to previous ones. This is a poor design decision as it removes the capacity for players to review lessons they have learned on previous levels or to try and improve their scores. Implementing this could be done by serialising their progress and previous scores, storing it as a cookie. This would allow both persistence in a player’s progress allowing them to leave and come back to their current game and, after the addition of UI control, allowing them to return to levels they have done before.

## 

# Assets

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **File Location** | **URL** | **Author** | **License** | **Accessed** | **Description** |
| /assets/bricks/ | https://opengameart.org/content/toy-brick-pack | Kenney | CC0 1.0 | 19/12/2019 | Set of Images used for the bricks. |
| /assets/images/bear.png | https://www.kenney.nl/assets/animal-pack-redux | Kenney | CC0 1.0 | 19/12/2019 | Image used for the muchkin. |
| assets/UI/kennyUIPack/ | https://opengameart.org/content/ui-pack | Kenney | CC0 1.0 | 19/12/2019 | Collection of images used for button backgrounds. |
| assets/sounds/click2.ogg | https://opengameart.org/content/ui-pack | Kenney | CC0 1.0 | 19/12/2019 | Sound used when buttons are clicked |
| assets/fonts/kenvector\_future\_thin.ttf | https://opengameart.org/content/ui-pack | Kenney | CC0 1.0 | 19/12/2019 | In game font |
| assets/fonts/kenvector\_future.ttf | https://opengameart.org/content/ui-pack | Kenney | CC0 1.0 | 19/12/2019 | In game font |
| assets/sounds/error\_sound.wav | https://freesound.org/people/distillerystudio/sounds/327736/ | [distillerystudio](https://freesound.org/people/distillerystudio/) | CC by 3.0 | 19/12/2019 | Sound played when block placement fails |
| assets/sounds/brick\_place\_pop.mp3 | https://freesound.org/people/greenvwbeetle/sounds/244657/ | [greenvwbeetle](https://freesound.org/people/greenvwbeetle/) | CC0 1.0 | 19/12/2019 | Sound played when brick or munchkin is spawned |
| assets/sounds/level\_complete.wav | https://freesound.org/people/LittleRobotSoundFactory/sounds/270404/ | [LittleRobotSoundFactory](https://freesound.org/people/LittleRobotSoundFactory/) | CC by 3.0 | 19/12/2019 | Sound played when level target is hit |
| assets/images/sky1.png | https://opengameart.org/content/sky-backdrop | Bart | CC by 3.0 | 19/12/2019 | Image used for the background |
| assets/images/target.png | https://opengameart.org/content/target | [Kyzerole](https://opengameart.org/users/kyzerole) | CC0 1.0 | 19/12/2019 | Image used as the level goal |
| assets/images/arrow.png | https://opengameart.org/content/glow-arrow | [oglsdl](https://opengameart.org/users/oglsdl) | CC by 4.0 | 19/12/2019 | Arrow image used to indicate munchkin spawn location |
| assets/UI/icons/ | https://opengameart.org/content/cartoon-mobile-rounded-icon-buttons | GDquest | CC0 1.0 | 19/12/2019 | Collection of images used for HUD button icons |
| scripts/phaserloader.js | https://github.com/johwiese/Phaser3-Loading-screen-asset-organization | Joh Wiese | MIT | 19/12/2019 | Used to load assets, modifications have been made to allow it to load bricks. |
| libs/phaser.js | https://phaser.io/ | Richard Davey | MIT | 19/12/2019 | Phaser game engine |
| assets/UI/HUDPanel.png |  | Jordan Harrison |  |  | Image for the background of the HUD |
| assets/UI/menubackground.png |  | Jordan Harrison |  |  | Image for the background of the menu |

## 

# References

Jakobsen, T. (2008) *Advanced*

*Character Physics*. Available at: <http://web.archive.org/web/20080410171619/http://www.teknikus.dk/tj/gdc2001.htm> (Accessed: 18/10/ 2019).

Liabru (2016) *How it Works*. Available at: <https://github.com/liabru/matter-js/wiki/References> (Accessed: 17/12/ 2019).

Löw, A. *How to create physics shapes for Phaser 3 and Matter JS*. codeandweb. Available at: <https://www.codeandweb.com/physicseditor/tutorials/how-to-create-physics-shapes-for-phaser-3-and-matterjs> (Accessed: 16/10/ 2019).

Riecken, J. (2019) *SAT.js*. GitHub. Available at: <https://github.com/jriecken/sat-js> (Accessed: 12/11/2019 2019).

wiese, j. (2017) *Phaser 3 - Loading screen & asset organization*. Available at: <https://jwiese.eu/en/blog/2018/04/phaser-3-loading-screen-asset-organization/> (Accessed: 20/10/ 2019).