* **Game summar**y (1 page): objectives, rules and gameplay.
* **Storyboard:** gameplay screenshots
* **Implementation specification** (1 page):

Provide a clear explanation of how the data structures and code used within the game are organised and operate. Document the overall flow of the code (high-level design / API) and the general code structure of your game. Here, you should explain how you implemented any major algorithms in your project. Do **NOT** include the extensive listing of the code itself.

* **Research (technical content)** (1 page):

Provide details of research that you undertook and explain how it influenced the game design and implementation. For example, you may wish to include details of an algorithm you used and explain why other alternatives would not work in your context.

* **Critical Review** (1 page):

Identify three reasons why the design and implementation of the application are good. Identify three areas where the game could be improved and a summary of how the improvements could be made.

* **Assets:**

Describe assets that are used within the game, particularly any assets that you created yourself.

* **References:**

# Munchkin Rescue – Report

Jordan Harrison

Contents

[Game Summary 3](#_Toc26963108)

[Storyboard 4](#_Toc26963109)

[Implementation Specification 5](#_Toc26963110)

[Research 6](#_Toc26963111)

[Critical Review 7](#_Toc26963112)

[Assets 8](#_Toc26963113)

[References 9](#_Toc26963114)

## Game Summary

Munchkin rescue is a browser based cross-platform game written in Phaser \*\*\* in which the player must attempt to get the munchkin to the goal in each level. The munchkin moves under the force of gravity and it is the players job to create a course it can traverse by placing blocks. Inspired by lemmings, angry birds and marbel 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.

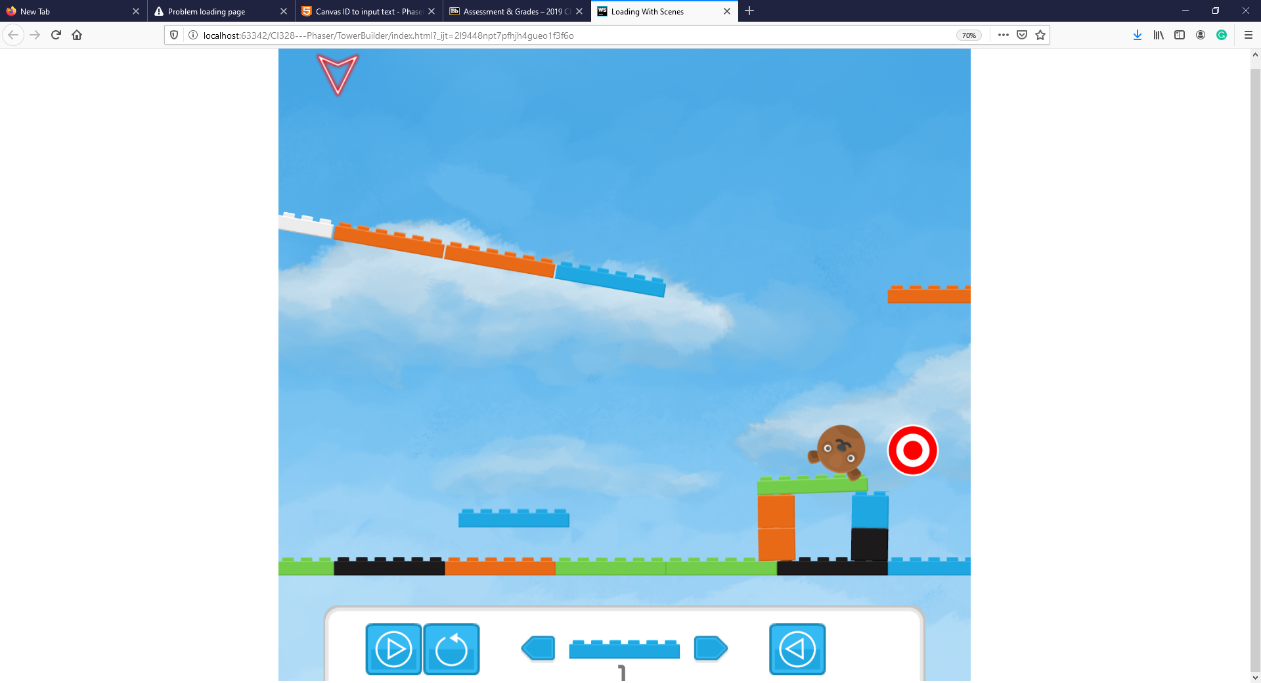


Figure 1: level 2 solution on easy difficulty

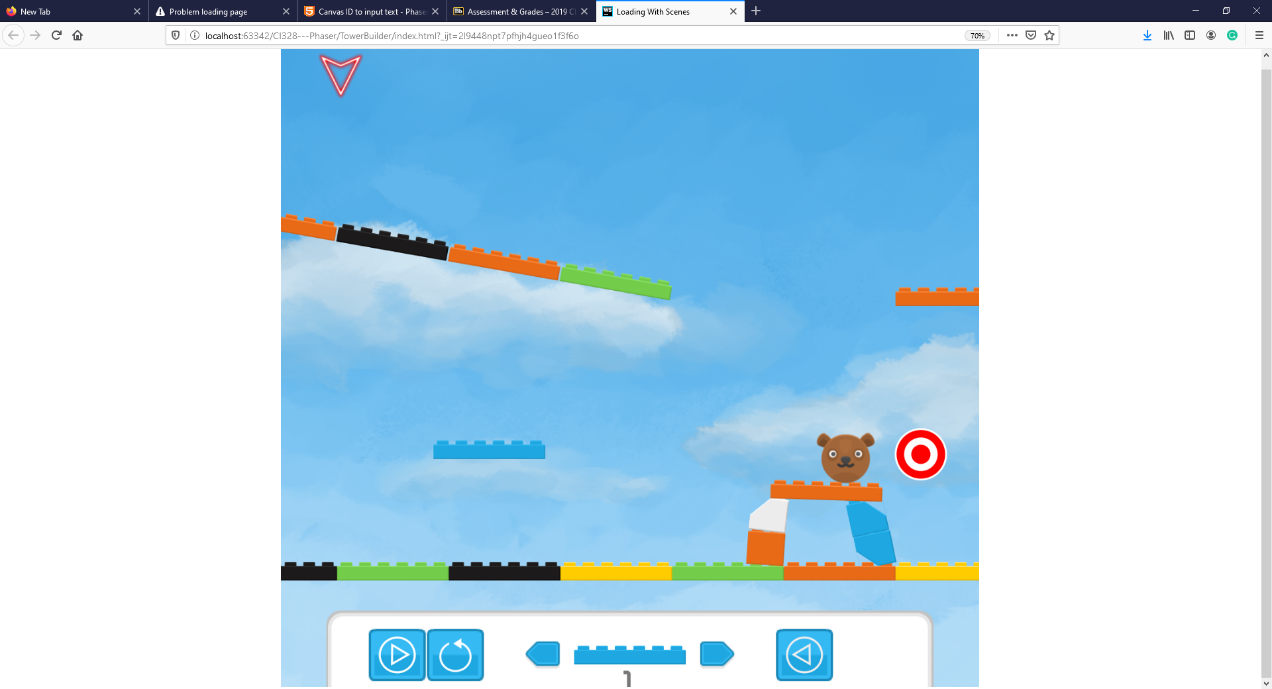
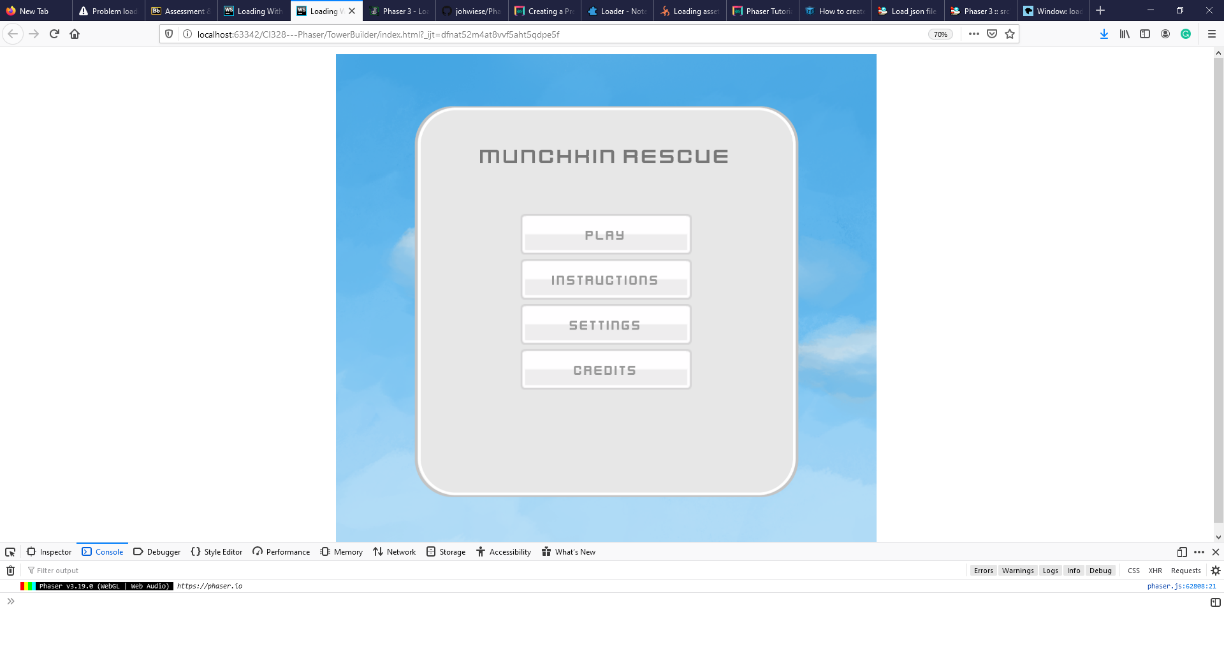


Figure 2: level 2 solution on hard difficulty

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



Player places blocks to help the munchkin travel to the target

## 

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

Level completes playing fanfare music and showers block particles

Munchkin successfully traverses the level to the target

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

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

Level objects are placed in and available blocks are configured

Game Finished loading and displays landing screen

## Implementation Specification

This first scene that is created is 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 allow 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.\*\*\*\*\*

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 done 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 enables 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 as 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 MatterJS \*\*\*\*\*\* 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.

## Research

## 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 enables the 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 lego 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 if 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.

There are two main issues with the scoring mechanic: there is limited feedback to the player of the result of their actions on the score and a limited diversity of the things that can increase or reduce their score. A UI element could be used to display the players current score, with changes to the score showing the affect actions have, audio ques could be used to enhance the feedback this provides.

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 players progress allowing them to leave and come back to their current game and, after the addition of UI control, allow them to return to levels they have done before.

## Assets

## References

<https://www.codeandweb.com/physicseditor/tutorials/how-to-create-physics-shapes-for-phaser-3-and-matterjs>

<https://jwiese.eu/en/blog/2018/04/phaser-3-loading-screen-asset-organization/>

<https://github.com/johwiese/Phaser3-Loading-screen-asset-organization>