AQA Computer Science
Non Examined Assessment

The Chrome Application Game Engine
github.com/ChilliByte/CAGE

Deep Sohelia
Center Number: 20153
Candidate Number: 8423

Contents

Overview:

A commonly occurring complaint of Chromebooks is that there are no games for them. Given the existence of Chrome Packaged Applications, the availability of Hardware-Accelerated Canvas and the advent of ES6, it seems that there should be a multitude of these games.

A browse of the Chrome Web Store shows many, many, games, however, the issue is that they are not well moderated and few still are fully packaged applications. The vast majority of these applications are simply links to websites, others are embedded flash games, and others rely on Unity, which is unsupported by Chromebooks. This lack of native, high quality games on the Chrome Web Store does not do the capabilities of the platform justice.

There would be more of these well-built packaged applications if there were an easy to use method of creating games, a Game engine, as it were. This way, games could be quickly written for the engine, packaged up and made to run on Chrome. This way, a game written once in HTML, CSS and Javascript can be run on almost any OS without having to be re-written or converted. The apps can even be made to run on Android natively, or even on iOS using Apache Cordova, although in order for this to be useful touch controls should be implemented, obviously.

I plan to write an 2D platformer game engine, written using ES6, the newest standard of Javascript which includes inheritance, classes and other object-oriented syntax missing from the old standard. It will be modular, so that functionalities for certain types of object can be added or removed, as needed. For example, if you require your Game to have water you can swim in, you simply need to include the relevant file, and if you need to remove the code for Ice blocks, simply dereference the file for that.

In order to showcase the engine and help guide its development, I will be designing it around a Game, that way the Game engine can be optimized for the intended game, rather than the end user needing to optimise their game code around my engine. It will be a 2D sidescrolling platformer, composed of 4 levels, 3 stages and a boss battle. The game will contain platforms, powerups, coins, ice, water, flying, trap doors, switches, low and high difficulty AI enemies, including bosses. All of these things would be modular, and ideally, would be downloadable from an online store, where people could share their add-ons. This, however is beyond the scope of this program.

Analysis

The Problem:

As mentioned in the overview, whilst the capability for fully packaged, well made games to be uploaded to the Chrome Web Store is available, it is currently too obscure and difficult to create and upload these, with multiple libraries needed to be set up and used simultaneously before a game can even be written. Due to this, most hobbyist or amateur game developers are still deciding to create their games with the older, less secure Adobe Flash platform, before hosting the game on a website, and simply adding a link to the Chrome Web Store. This may not seem an issue, as Chrome can be made to open these applications in an application window, issues arise when these websites use libraries not supported by Chromebooks, the key users of the Web Store, or, more menacingly, when these websites spoof Google login pages to phish player's login data before redirecting to a low quality flash game.

Example 1: Earth Keeper 2

Figure 1: A screenshot of Earth Keeper 2 running on Chrome OS

As you can see, the game is running in an application window by default, and there is no option to let it run in the browser, this shows that the Game is in fact a packaged application. It is lightweight at 16mB, and runs very well, despite my Chromebook only having a mobile processor. It runs consistently at 60 frames per second.

Figure 2: Screenshot of FPS meter

However, the game was written in Scirra's Construct 2, a game creation program for Windows and OSX. This severely limits Chrome OS or Linux users, and not many functional game editors exist for these operating systems, especially in the case of Chrome OS. To develop games for Chrome OS, it would be most useful to be able to develop those games on Chrome OS, preferably for free, so that game development and design is accessible to all Chrome OS users.

Example 2: Tribal Wars 2

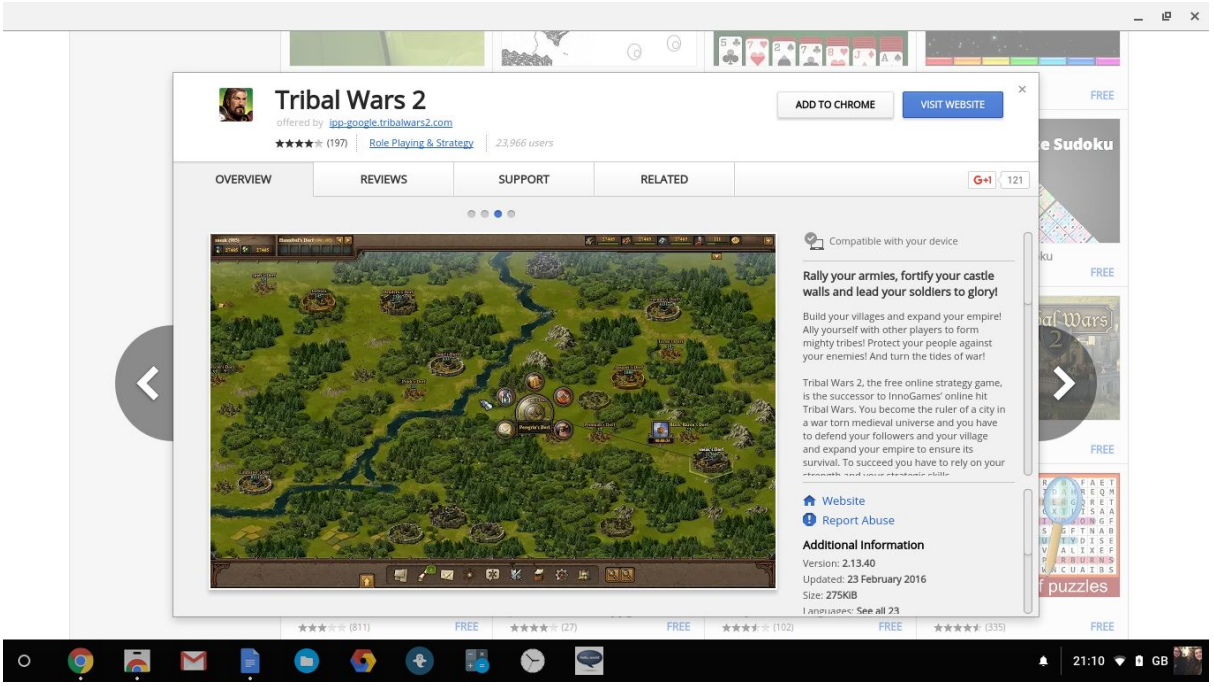


Figure 3: The download page for Tribal Wars 2

Most games on Web Store are more akin to this, a Web-Based game, with a corresponding Chrome "App" that merely opens the game in a new tab. The games vary in quality, however Tribal Wars 2 is very good compared to the rest of the games in this Category, most of which are of a low quality, either being uneventful and tedious, or downright dysfunctional clickbait. These are normally written with HTML, CSS, JS, and Node.JS, though the lower quality games are often written in Flash, before being deployed onto the web. After this, the creators quickly create a Chrome Application and add it to the Web Store. This may also be generated by a program if the game was written in a software package that had this functionality.

The Current System

Most games are currently made in a drag and drop, graphical program, like the aforementioned Construct 2, by Scirra. These programs contain a user interface in which to design and develop the game, as well as a way to demo the game, and programs to export the game into a relevant, or many relevant, file format.

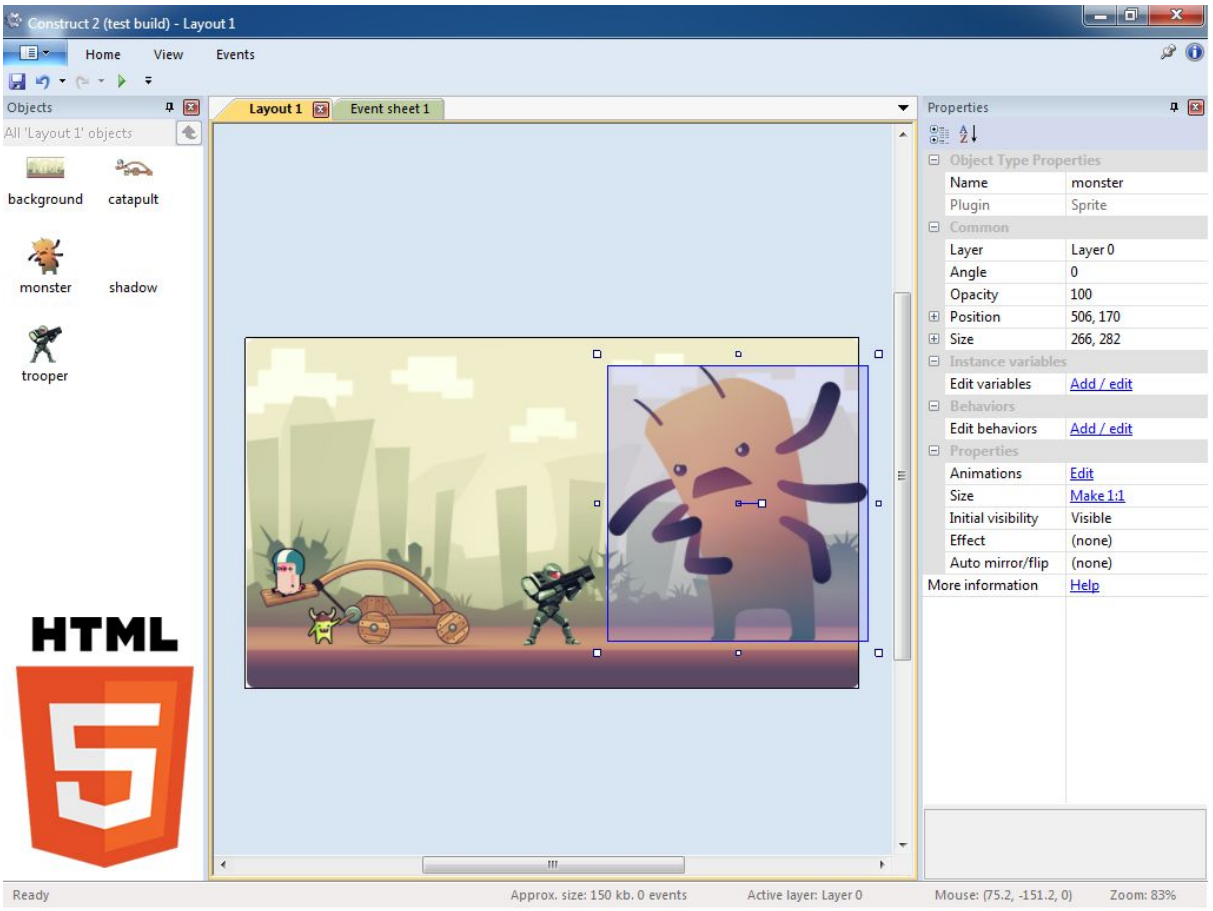


Figure 4: A screenshot of Construct 2, editing the demo game provided with the package

In Figure 4, you can see three panes, on the left, there are the objects, which are entities such as players, enemies, pickups and platforms for use within the game, which are not critical to its functionality but are to its aesthetics. In the center is the stage, where the game elements are arranged and positioned. This is where the physical level is built: the way the blocks are placed, where enemies appear, where powerups are available, et al. On the right are the properties of selected objects, in this case, the large enemies, here you would be able to edit the properties of the object, such as its size, its maximum speed, and a whole host of other properties, the list is limitless.

A basic level may be composed of three objects: A platform, a player, and a goal. These three would be created using the relevant menu, and dragged onto the center stage. The rightmost menu would then be used to set the platform as an unmoving solid, the player as a solid controlled by inputs, and the goal as a fixed non-solid, which would

reset the game when hit by the player. This is the most basic of games, and more complex behaviours can be created fairly easily. How to do this, of course, is beyond the scope of this document.

As a Chromebook user, and a programmer, I have always wanted to create games for my Chromebook, but due to the limitations of the Operating System, packages like Construct 2 cannot be installed, and a web-based alternative must be used. In fact, this is such a frequently asked question, Google has written a page addressing it (https://developer.chrome.com/apps/game_engines), where they recommend PlayCanvas and WiMi5 as web-based methods of creating a game.

However, these both have their own limitations. PlayCanvas is very feature rich and is very professional, however it struggles to run on my machine, which is one of the better Chromebook models, meaning that to run PlayCanvas well, you need either the more powerful Chromebook Pixel, which costs £800, or a Desktop PC with more power, both of which defeat the point of using the budget-price Chromebooks.

WiMi5 suffered the same issues as PlayCanvas, but on top of that, instead of a simple property editor, uses a visual programming language, which even after 3 hours of trying to decipher, I could not use to create a game more complicated than the tutorial game. Any solution I create to this problem should take this difficulty into account.

Proposed Solution

In order to remedy these poor quality games, and this lack of a method to quickly develop enjoyable games, I wish to develop a simple Javascript Based Game Engine, which can be used in conjunction with the Chrome OS IDE, the Chrome Dev Editor (or any other IDE of the users' choice) to develop a game. I want the system to be flexible and adaptable to any type of game, and as such, it should be a modular system. It should take in a set of module files, image files, and level files, and create a fully functioning Chrome Application complete with menus and progress saving.

Ideally, the Engine and Modules would all be downloadable from one organised, centralised, resource, such as a website, to which people could add their own assets, modules, levels or even games. However such a system would be very complex to set up, and would require a large selection of modules to begin with in order to build a user base. A more achievable goal for this project would be to write for core functionality of the Game Engine chiefly its modularity and its ability to output as a Chrome Application.

Prospective Users

The prospective users of the game engine will be novice programmers who want a clean, easy way of learning how to create a game on any device. The physical game will only be a proof of concept, and as such will be targeted at people roughly my age who have played many games and will able to give valuable feedback on intricate details that

others may miss, which will help me improve the engine. I will be testing and receiving feedback from Computer Science students of varying technical ability, as they are of the skill level my game engine is designed for, and as such they are an easily accessible subportion of my target market. The names and backgrounds of the prospective users I have gathered are:

Name	Years Programming	Background
Hugh Wells	8	Backend Web Engineer
Matthew Randell	7	Various
Sam Needle	1.5	Comp. Science Student

User Needs

In order to create enjoyable, high quality games, a Game Engine that is capable of creating them. As such, I need to create a system that is capable of:

- Creating Chrome Applications
- Creating and managing Game Objects
- Running and Displaying the Game
- Taking in custom level files
- Taking in custom module files
- Supporting multiple devices

This will likely be done programmatically instead of graphically in order to keep the system fast and lightweight. Due to this, a comprehensive set of documentation will be needed along with commented code in order to make the system understandable and easy to use.

In order to facilitate the development of better games, the system needs to be able to:

- Handle many diverse object types
- Handle many levels at once
- Handle a large variety of media, such as sprites, backgrounds and music
- Generate lightweight applications

Aims And Acceptable Limitations

The system is severely limited by the devices it is running on, and as in theory, its scope is limitless, it is important to precisely define what is an is not within the scope of this project.

Aims:

- Create an ES6 based modular 2D game engine
    - This will be done by writing a series of scripts which will handle creating a Chrome Application, as well as setting up a Game Canvas which can be accessed, modified and drawn to
- Create a set of demo modules to work with the engine
    - These will include core modules that give the key functionality of a platformer game, as well as a host of ancillary modules which will extend the functionality of the engine.
- Create a basic game as a proof of concept of the system working
    - This will be a small platformer, with a small handful of levels and a sandbox level which will be used to demonstrate all of the features and modules I have written in action

- Create a centralised resource for the game engine, containing tutorials and documentation on each of the modules I write
    - This will be written in Markdown using GitHub's wiki feature

Limitations

- Ultimately, I intend for the system to be collaborative, and as such, an exhaustive list of modules is neither practical nor necessary, and I should only create a small number as a proof of concept

- Building on the above point, while a collaborative system for assets and modules would be required for the system, it would be a very complex and intricate system, more so than the game engine itself, as I would need to handle user accounts, build version control systems and content delivery networks etc. Such a system deserves its own project and in depth analysis, and as such, it is beyond the scope of this program. However, a centralised resource is still needed, and that is one of my aims.

- Chromebooks struggle with 3D graphical rendering, and as such 3D Graphics are beyond the scope of this program.

- I am not a Game Designer, nor a Graphics artist, and as such, the sample game does not need to be of a high quality, as the project is focused on the engine and its capabilities to produce a high quality game, rather than mine.

- A graphical level editor, while useful, would be difficult to run on Chrome OS, and as such would not be feasible.

Justification of Solution

The key features of the solution I have proposed are that it is written in ES6, modular and collaborative. These are all key features if I am to begin tackling the problem of low-quality games on the Chrome Web Store.

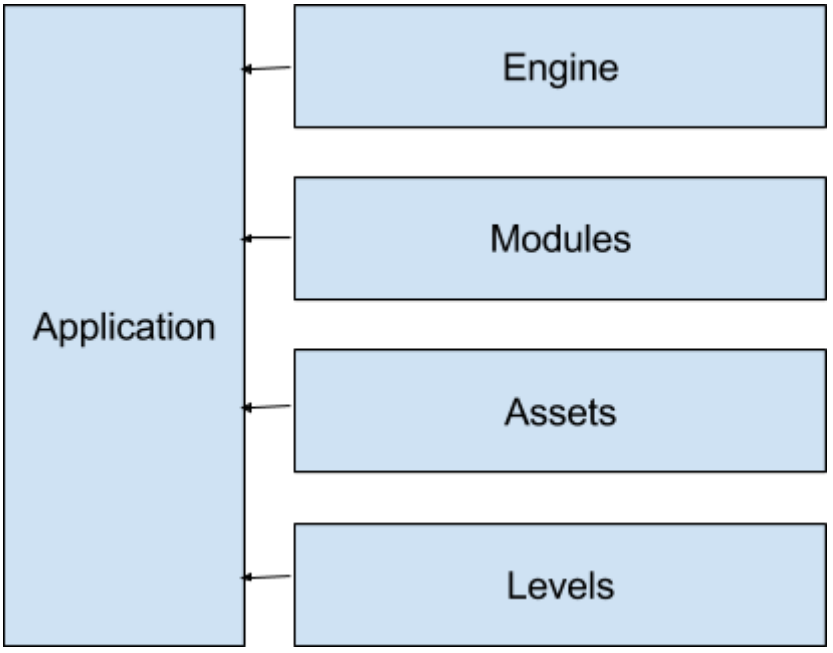
The use of Javascript is mandatory as it is the language with which Chrome Apps are written, and ES6 is the newest Javascript (or ECMAscript) specification, and includes Classes, inheritance and Arrow functions, equivalent to lambda functions.

Modularity is where the system gets its power and flexibility. I could write the system to contain every type of game object conceivable, but almost all of these would be unused - your platformer would never use a set of tile-based enemies etc. Modularity means that the game will only contain code relevant to it, and no code will go unused. This way, games are kept lightweight, but the system is still adaptable. Instead of newcomers attempting to modify the game engine to implement a feature, they can merely import a module from a centralised resource to add the functionality they need. The system should be able to work be the game a turn based card game or a multiplayer platformer.

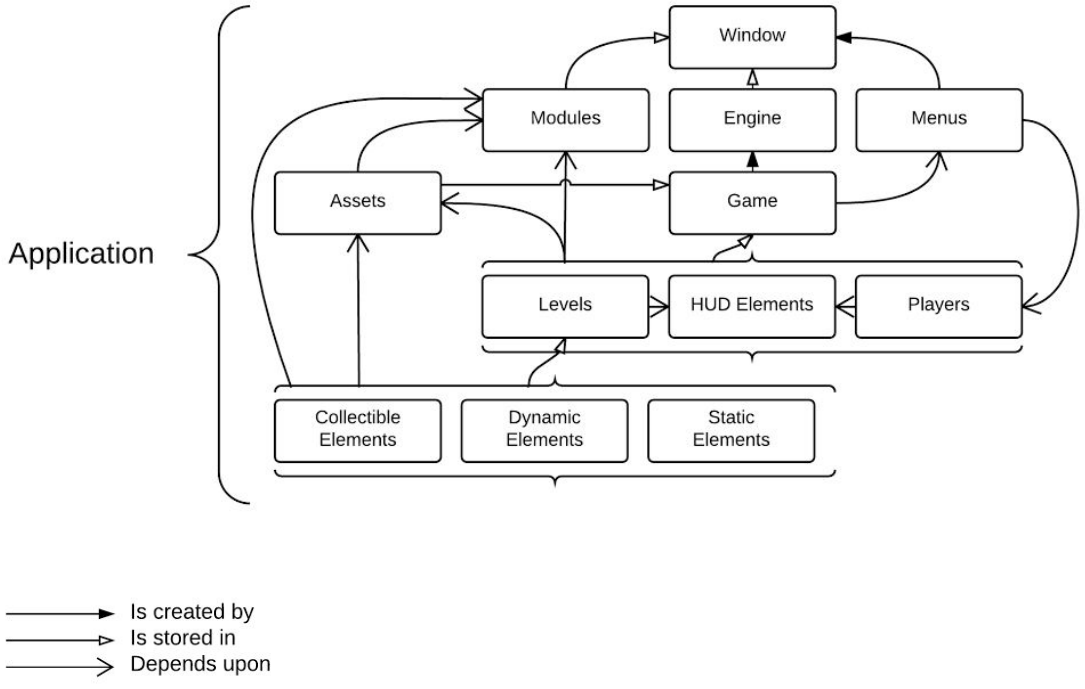
Of course, for the system to be adaptable and modular, there needs to be enough modules to facilitate this, quite literally hundreds, and as such, it would be impractical for them all to be written by a single person or team of people. Ideally, the modules would be written by people who would upload the modules to a central source, for other users to incorporate into their games. This way, there would most likely be more modules that if they were all written by myself. The more modules there are, the greater the range of usable objects within the produced games, and the greater the range of games that can be produced.

Proposed structure of new system

File Structure:



Game Structure



Design

The design process consisted of multiple stages, beginning with interviews with my prospective users, then leading to more in depth definitions of the structure of the game engine, how the modules work together, the program flow and the full list of modules I will create.

Interviews

I began the design process by speaking to my prospective users about what they would require in the system, and used those to form a more thorough design specification to build upon my aims. Sentences I have said are in **bold**.

Q) What would you say is the most important aspect of a Game Engine?

Hugh:
Game Engines in general? Or just yours?
In general.
I think it's important that they are lightweight, fast and widely supported, especially if they are being used to create games that are ported across multiple platforms. I think a game should be a standalone application and I really dislike having to download other programs or plugins to make my game function. And because of this, I think that the game engine should be able to produce these standalone games that use technologies that are widely supported. The speed and filesize I mentioned are sort of generic, frankly, I think all games should be as small and as fast as possible.
You said technologies that are widely supported, what does that include?
Anything written in C or Java, those are supported by most devices. I think web technologies are the most widely supported, I can't think of anything that doesn't support them, and it would mean that you wouldn't have to mess around with porting your games.

Matt:
That's quite a broad question there, Deep. Flexibility, I think, is the most important, I should be able to use the same game engine to create a puzzle game or a multiplayer action game.
Would you say a modular structure is the best way to accomplish this?
So you mean like I could add in or remove game elements as necessary?
Yes, so you could simply include a "platformer character" file as well as a "platform" file, and then use those within your code, or equally a "gameboard" file and a "game piece" if you wanted to create a board game.
Yeah, I like that idea, it would make the game engine very flexible. Though, I think it'd be important to have a lot of those modules, and make them easy to implement, else your system becomes difficult to use.
How many would you say is sufficient as a proof of concept?

I think that depends on how you're going to show off the engine.
I'm going to be creating a simple platformer game with about 4 levels
In which case, I think about 15-20 should be enough, given that about half of those will probably things such as a level module, or an input module.
So excluding modules necessary to set the game up, you think 5-10 extra modules is about right?
Around that, yes.

Sam:
I've never really used a Game Engine. I play quite a few games, but I've never really made one. I think ease of use is the most important features, I don't particularly want to spend hours reading up tutorials on how to use a game engine. I want to be able to download it, install it, and get to designing levels almost immediately.
Would some form of "getting started" project help with this, so that as soon as you open the program, a simple 1-level game is waiting for you to modify?
Assuming the interface is intuitive enough for me to modify levels without too much help, I think that would be the best solution, as it would save me from having to set up all of the game settings or whatever each time, I could just modify a preexisting game.
You said "intuitive", could you elaborate a little on what you mean by that?
I'm not quite sure what your final interface will look like, but what I mean is that I should not have to read up on the fine details of your engine to perform basic actions. If I want to create a new object, the way I do that should be as few steps as possible, and as logical as possible.
Would you say a "new object" button/function is intuitive enough?
If it's logically named, yes, I don't want to be guessing what things do.

Q) Are there any key features you would like to see in my engine?

Hugh
If this is a Chrome Application game engine, you really need some simple way of actually creating the application, without the user needing to be an expert with chrome.
Most of the app generation is handled by Chrome, if the app is written in HTML etc, all you have to do is upload it.
Ah, that sounds reasonable, just make sure that is made clear to the user, as they will undoubtedly want to play their game and I'm sure they won't all know how to find the upload button.

Matt
I think the way you handle HUD elements and menus should be simple and flexible, users should be able to create these elements without needing an in depth knowledge of whichever graphical library you use.
If I were to use Web Technologies, would you say writing these elements in HTML and CSS, then adding interactivity through Javascript would be too complex?
As long as the system was logical and you provided sample menus and HUD Elements which would show how they would be created, I think that'd be acceptable. Novice game designers shouldn't have to learn ES6 just to create a "play" button.

Sam
I think an easy way of creating levels is the most important, I should be able to place game objects precisely where I want to without any difficulty. That, and a wide range of game elements so I'm not limited in what I can produce.

Q) Are there any advanced features you would like to see?
Hugh
If you're going to be using web technologies, some form of integration with WebSockets to facilitate multiplayer between computers would be pretty cool.
Do you mean real-time or turn based multiplayer?
If it's feasible, both, though I suspect that might be beyond the scope of your project.

Matt
I'm assuming you're going to add some form of AI? If you're going to have platformer enemies?
Yes, though they'll likely be simple, like a patrol between 2 points sort of thing.
Maybe some way for the more advanced programmers to create their own behaviours? Like maybe add a boss fight as a way of showing how custom behaviours may be defined
Would you say these should be modular as well?
I think for this project, a single, hard-coded boss battle should be sufficient, but if you take this project further and add multiple bosses, they should be contained to their own modules so that players can add and remove them as they please.

Sam
Advanced features? What like?
Well, two features that have been proposed is real time cross platform multiplayer, or custom bosses.
Oh, so like things that would require the game designer to know how to program?
Yes, more advanced features that aren't necessarily transferrable between games.
I'm not too sure. Maybe some way of having different level types in a game?
What like?
I don't know, let say I was building an open world isometric game, I would have a few "isometric level"s would I not, for each town or whatever
Yes.
And if I wanted the routes in between the towns to be sidescrolling levels, I would have to add "platformer levels" to the game
Yes, and you're suggesting a way of handling both these types, and the different physics between both would be useful?
Yes, either have this all handled by the game engine itself, or allow me to detect my own custom game events.
Game events?
Yeah, so I could write a function to check if the player has stepped on a platform, and if they have, then summon in a new enemy or something. And that code would be run every frame so I never missed anything.

Q) What are your feelings on a text-based way of creating levels, as opposed to a GUI?

Hugh

I'm not sure, It sounds like it could be a little cumbersome to use, especially if you can't see your level as you edit.

If your game updated instantly as you edited, and all you had to do to view your changes is refresh the game, would that be acceptable?

As long as there was some way to skip to the relevant part of my game, yes, I don't want to have to play through all of it to view the last level.

Matt

I'm not quite sure what you mean. Would users be typing numbers into a 2D array? Would they be writing an XML file? JSON? I need more information.

I haven't finalised the format yet, but it would likely be based on ES6's constructors.

How would those constructors be used?

So a user would define a level variable, and then add objects to the level using some form of level.add() method, and the object to add would be generated via a constructor function.

So you would be creating game objects, and then be pushing them to a game level?

Yes.

I think that's logical, though the constructors should be logically named, and a list of them and their arguments should be written up somewhere in order to make them easier to use,

Is the source code an appropriate place for this?

Yes, but you should Ideally have some form of documentation as well.

Sam

I don't see anything wrong with a text-based level editor, but why wouldn't you just write a visual one?

Chromebooks aren't very powerful, and I want this system to work on Chromebooks, Raspberry Pi's, anything really, regardless of power

Ah, I see, well, as long as I have a simple method of creating levels and objects, and I can also view my level as I edit it, that would be fine.

Other people I interviewed also raised those points. Would you say they are absolutely necessary, or simply an added extra?

I think they are necessary in order to actually create a game, else you would be the only one who would know how to add objects to your game.

Q) Is documentation important to this project?

Hugh

Almost certainly, you need to write up how every single piece of your engine works, as someone will inevitably have an oddly specific thing they wish to implement and they will need to know exactly how your engine works.

Matt

I think tutorials would be more appropriate than traditional "this function does this and takes these parameters" style documentation. I would probably write a few tutorials and then simply document what each module does and how to use it, without going too much into specifics.

Sam
I have no idea how your system will work, and neither will I know how to use it. You absolutely need documentation. Combined with the demo project i mentioned earlier, you should probably write up what each module does, and how to use it. The specifics of how it works should probably be written up in the source, but not in the documentation

Game Engine Structure

The game engine would likely be composed of five different sets of files:

- Chrome Application Files
- Game Engine Files
- Modules
- Levels
- Assets

Chrome Application files are files that are required for the Chrome App to function, they include:

- A manifest.json file to specify data about the app, such as its name and version
- A background.js file to launch the app
- An index.html file to display
- And a set of icons in an assets folder

The game engine files would handle creating a canvas, and setting up key game objects and variables, before the modular elements would add the functionality to the game canvas.

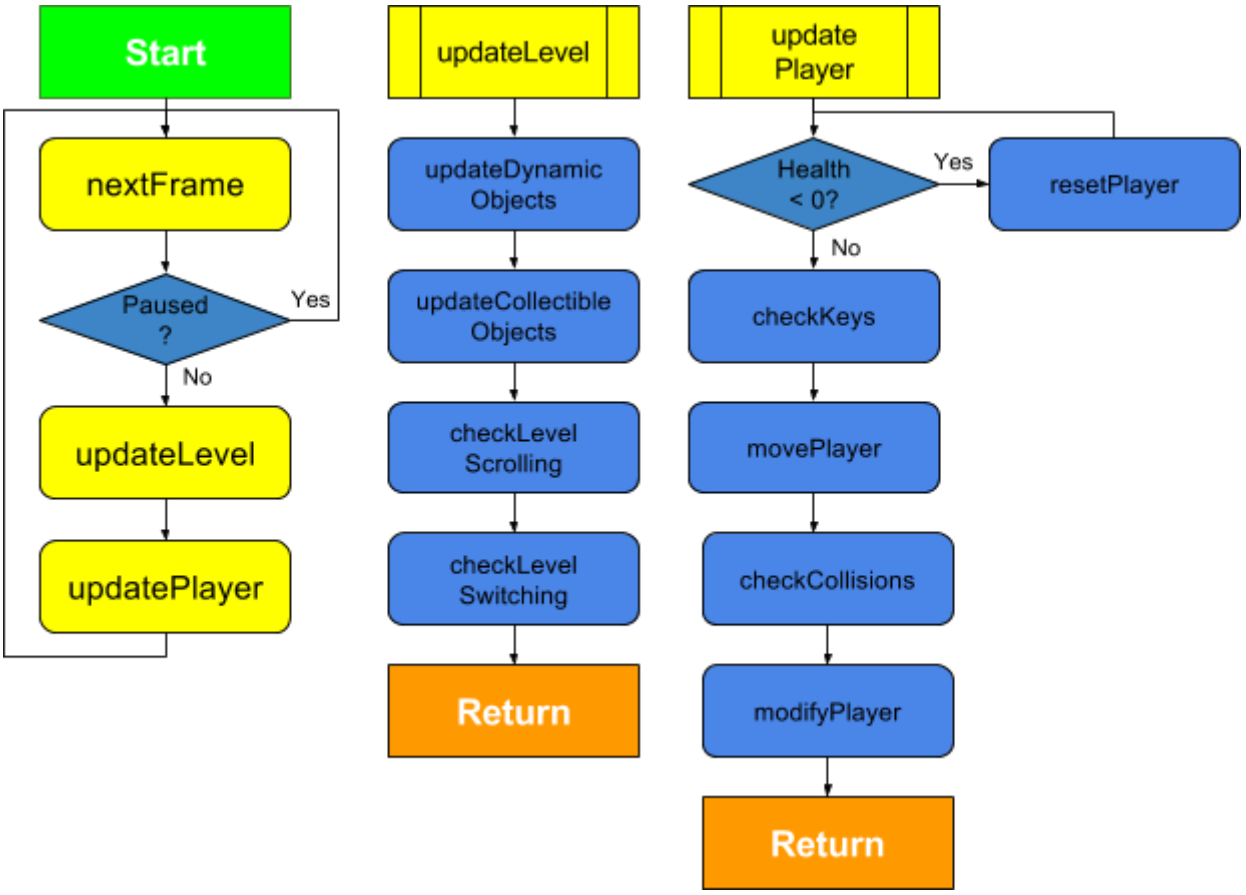
The modules come in two classes, core and ancillary, with core modules adding key functionality to a game, such as levels, players, user inputs, and ancillary modules adding extra features, such as powerups or special types of enemies

Level files are written using methods and functions defined in the modules, and are included in the index.html file along with the modules to load them into the game.

Assets are various types of images that are are used in the game, such as backgrounds and sprites.

Game Flow Structure

The game would likely consist of a series of function calls and iterative loops, nested within each other in order to provide the functionality of a game.



The game loop would run every frame, at roughly sixty times per second, and that should update the game each time it is run, if the game is not paused. The above diagram roughly shows the sequence of events the code would follow in doing this.

The level is updated first as elements of the game (enemies, moving platforms etc) will need to move and update the status of anything they collide with. The level may also need to updated to scroll it along or switch level. If the scrolling was done after the player was updated, the level would judder between frames as the player would move, then the level would scroll the player back next frame, and then this would repeat.

Once the level has updated, then, the player can be updated. First the user's key presses are checked, and these are used to update the player's velocity. Then, the player is moved depending on its velocity. The player is then checked for collisions between all game objects, to make sure the player interacts with them. The actual interaction is handled both by the player, and by the object it collides with

Module Structure

Modules would come in four varieties:

- Statics,
- Dynamics,
- Collectibles
- Unclassifieds

Static objects do not move relative to the level, and would not interact with any moving objects, beyond collisions. They would not be updated every frame, and they do not need to be reset. The most common type of static module would likely be Platform

Dynamic objects are any objects that change state between frames, or that need to be updated when collided with, Enemies, Moving Platforms and Switches are all examples of this. Dynamic objects are checked for collisions between all other objects.

Collectible items are dynamic objects which can be collected, and which only interact with the player. Coins and Powerups are the most common example of this

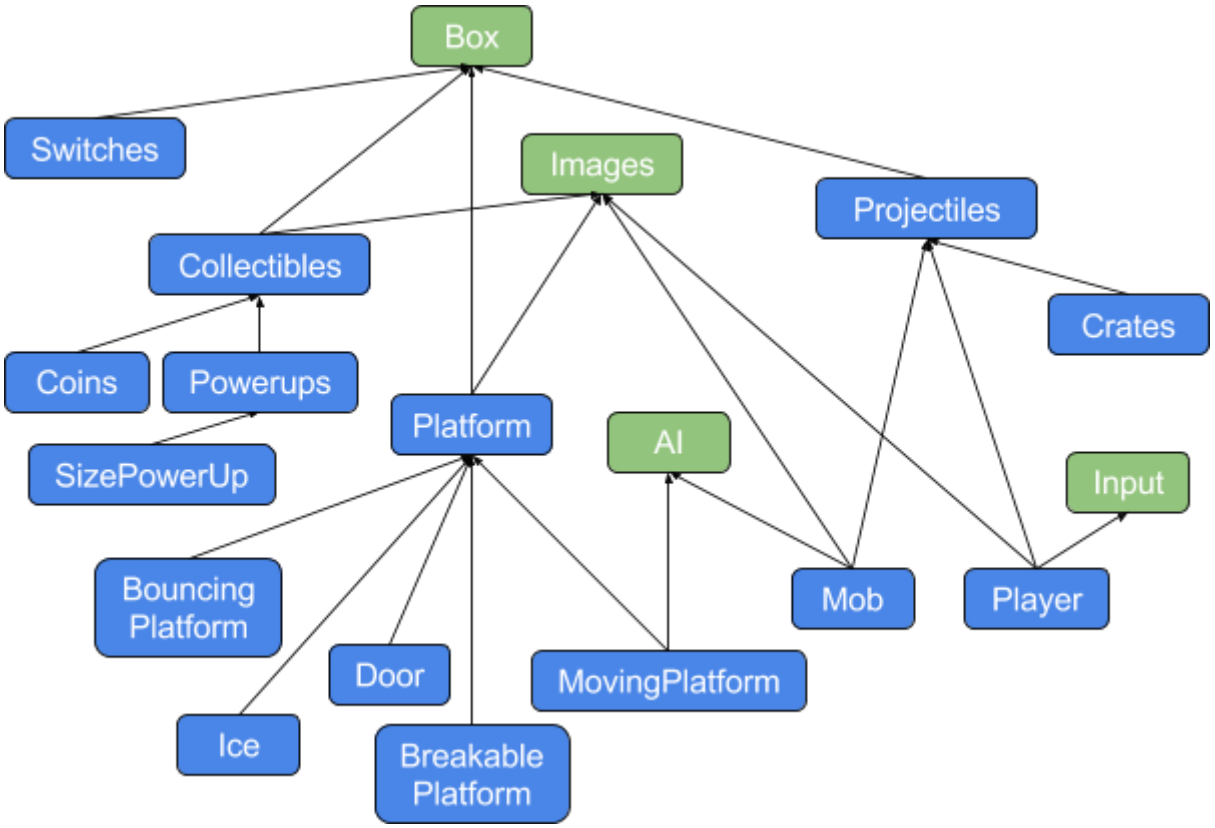
Unclassified modules are modules which are not game objects, but are required for the game to function. Levels, Images, AI, are all examples of these. All static, dynamic and collectible objects can be pushed into a level, Unclassified modules cannot be added to a level. Players, therefore are Unclassified objects, as they are not pushed to a level, they are pushed to the game object, and they are drawn within a level.

Full Module List

Modules:	Core	Ancillary
Statics	Box Platform	Doors Ice
Dynamics	Projectile	BreakablePlatform BouncingPlatform Crates Mob MovingPlatform Switches
Collectibles	Collectible	Coin SizePowerUp
Unclassifieds	Level Player Input Images	AI

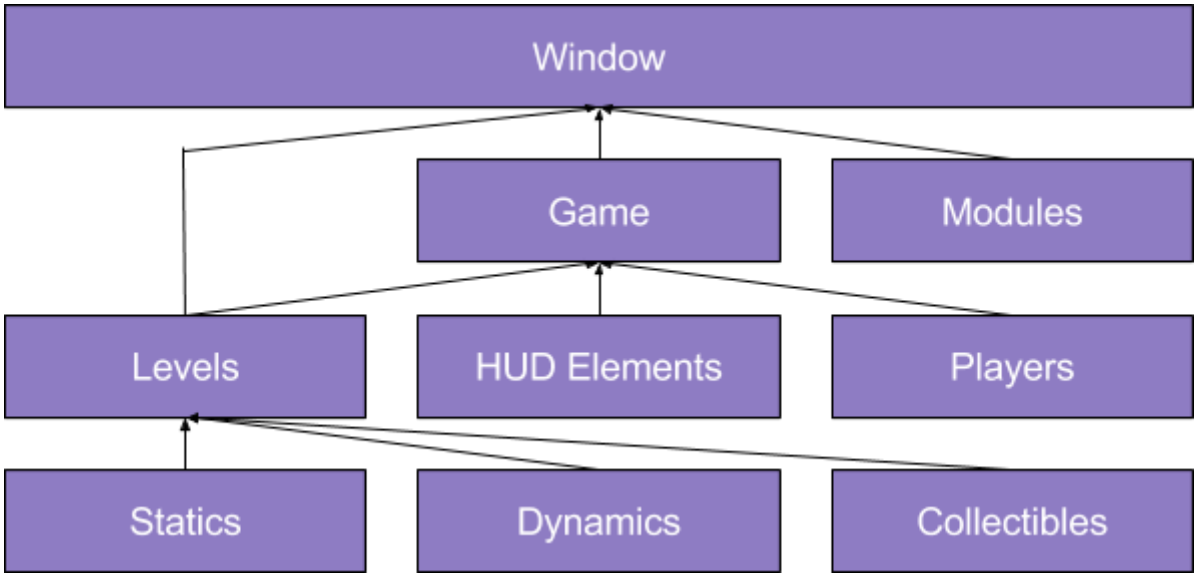
Module Inheritance Diagram

Green boxes indicate root classes, blue modules have dependencies



Full Program Structure

The elements of the game will be pushed into a hierarchical structure when the game is loaded, in the following structure:



The game object, modules and any defined levels are pushed to the window object when they are defined, as they are global entities. HUD Elements and Players, along with all of the levels, are pushed to the Game Object, and all of the classified game objects are pushed to their respective level objects.

This diagram shows the actual objects, the instances that are loaded while the game is paying, and should not be confused with the inheritance or file structure diagrams shown earlier.

Final Aims

My initial pre-design aims were:

- Create an ES6 based modular 2D game engine
- Create a set of demo modules to work with the engine
- Create a basic game as a proof of concept of the system working
- Create a centralised resource for the game engine, containing tutorials and documentation on each of the modules I write.

I am modifying these aims, given what I have discussed with my prospective users, and given my designs, to add the following:

- Create a ES6 inheritance based module structure, utilising polymorphism and aggregation to create small, lightweight modules that can be used to add functionality to games.

- Create a sample, single level project aside from the basic game as a project to demonstrate how to create and edit levels
- I will fully document every method and property of every core module in order to make the system user friendly and accessible.
- I will add a method of detecting when certain events have occurred in order to trigger certain user-defined behaviours.
- Write a set of logical constructors for the users to use when creating levels.

Technical Solution

In this section i document every module, how it works and what its use is. Areas of note, where I use especially interesting or complex programming constructs include:

- The update() functions of game objects, which are examples of polymorphism
- background.js which, combined with main.js, governs the Chrome App.
- The Ancillary modules, which demonstrate modularity, and inheritance.
- The player module, which highlights the use of more advanced OOP techniques such as static methods.

Full code can be found in the appendix.

How CAGE works:

CAGE is a modular Game Engine, and is composed of two different types of script: Core Engine scripts and Modules, upon which your levels and assets will function.

Core Files

CAGE is composed of multiple core Javascript files which work together to create your game. These are:

`index.html` `style.css` `main.js` `game.js` `background.js` `manifest.json`

These will set up a Game Stage for you to add objects, players, enemies and other modules onto. I document these files further below.

Modules

These are scripts, such as `box.js` or `movingPlatform.js` which add functionality to your game. Some of these are more useful than others, and some build on the code of others. For example, `breakablePlatform.js` builds on the code for `platform.js` and so on. If you want to add a module to your code, it is important to include all of the modules it relies on. These modules are stored in `/modules`

Level Files

Once the Core Files have set up the game stage, and the Modules have been imported to add the required functionality to your game, Level files are where the stages are built, and the magic happens. The modules add **constructor** functions, which let you add objects to your game. For example:

```
level1.add(new Platform(10,5,6,1,Red));
//Adds a 6x1 Red Platform at (10,5) to Level 1
//*Red* is a an instance of BlockColorTile. See images.js for more details
```

Images

`images.js` and `sprites.js` deal with the way images are rendered to the screen. `images.js` contains the code to draw the images, and `sprites.js` is where the sprites are defined. For example, you may define a `DirtSprite` to use when making platforms. This would be done in `sprites.js` and is documented below.

background.js and manifest.json

These files are for Chrome, and they store data on what the app is and how it should run. Users should not usually need to touch `background.js` unless their game requires multiple windows or other advanced features. These files are based off the sample manifest and background created by the IDE I used, the Chrome Dev Editor.

They will, however, need to change manifest.json to the name of their game, and its current version. The properties for this are listed below. More experienced users may wish to change the icons or even permissions to suit their game, but this is not usually necessary.

```
"name": "Platformer",
"short_name": "Platformer",
"description": "",
"version": "0.0.1",
```

index.html and style.css

Index.html is used to load all of the relevant modules and levels into the game, as well as to build the menu elements.

index.html

To include a script file, be it a module or a level, the javascript file must be included, like so:

```
<!--In index.html-->
<script src="pathTo/yourFile.js"></script>
```

Scripts are loaded sequentially from top to bottom, so included files must be included after any modules they rely on, and before any modules that rely on it.

Menus are also written in this file, which i document later.

style.css

This file is used to style the game and menus, and is not in anyway different to a normal website stylesheet. There are a few styles in the default provided stylesheet, which handle positioning and resizing the game canvas, as well as a few basic menu styles, but users are free to write their own in order to personalise their game.

main.js

main.js contains the key game variables, code to start the game once the page has loaded, and code to set the size of the Chrome App Window. Most users will only need to use this to set their

Key Variables and Definitions:

```
canvas, //Stores the HTML Canvas Element
c, //The canvas context, used for drawing
tilesX = 40, //The number of tiles across the screen
tilesY = 20, //The number of tiles vertically on screen
pixelsPerTile = 100, //The number of game pixels per tile, more details below
u = pixelsPerTile, //u is used as a shorthand to save space.
currentTime = 0, //used for delta timing, if you wish to use it.
oldTime = 0, //currentTime and oldTime both store unix timestamps each frame,
with oldTime storing the previous frames timestamp.
delta = 0, //stores the difference between the currentTime and oldTime
keys = [], //Array which stores which keys are being pressed with a boolean
modules = [], //Array to store which modules are included, used to check if
dependencies are met
totalModules, //Used to make sure all modules are loaded correctly
currentLevel = 0, //Used to store the currentLevel
totalLevels, //Used to check all the levels have loaded properly
```

Most games will almost certainly require more variables than this, and the default main.js comes with extra variables to work with the default modules.

Game pixels, mentioned above, are independent of screen size, to make sure the game runs correctly regardless of the screen size. The game, by default is 4000 by 2000 pixels, and it is then resized by CAGE to the size of the player's screen. This way, if your game moves the player by 1000 pixels to the right, the player will always move a quarter of the clients screen width, rather than miles off the page if they are on mobile, or an eighth of their screen if they have an 8K monitor.

Functions

See `/application/main.js` for code

`window.onload()`

Runs `reset()`, fetches the canvas, gets its context and sets its size. Then calls `render()` (See /application/game.js) or game.js wiki) if the level files are loaded correctly. This is done via `scriptCount()`

`reset()`

Moves the chrome window to the top left corner, and sets it to max width and height If you wish to adapt CAGE for a webpage, simply remove this function, though this is `not` recommended

`scriptCount(type)`
Loops through `index.html,` and counts the number of included scripts in the folder "type", and returns it.

game.js

Game.js contains the Game object, as well as the main render loop, which is run every frame and starts adding to the call stack

Game Object

The game object contains important properties and methods for the game to run

Game Properties:

`paused debug levels players hud`

`paused` and `debug` store whether the game is paused and whether the game is in debug mode. `levels`, `players` and `hud` are arrays which contain the relevant objects. the `Level`, `Player` and `HUD` modules should push to these arrays when a new object of one of these types is created

Game Methods

`pause() play() kill()`

These should be self explanatory, but `game.pause()` temporarily pauses the game, `game.play()` plays the game, and `game.kill()` redefines the `render()` function so that it does nothing

The Render Loop

This is a loop of code that is run every frame, which tells the game to update itself and to draw itself to the canvas, amongst other things.

First, it generates a multiplier based on the difference in time between frames. I set this to 1 for my demo game, but to enable delta time, just delete the 1 and uncomment the code.

```
currentTime = new Date().getTime();
delta = currentTime - oldTime;
oldTime = currentTime;
multiplier = 1 /* delta/16.6 */ ;
```

Delta time is the way to ensure your game runs the same no matter the speed of the machine. If the machine is slow and the game runs at half the normal frames per second, the change between frames needs to be double to account for this. This multiplier is passed onto the game for it to use.

Then, the render loop updates the game, if the game is not paused:

```
if(!game.paused) {
    //First update all of the game elements
    game.levels[currentLevel].update(multiplier);
    Player.updateAll(multiplier);
```

`LevelObject.update()` and `Player.updateAll()` are documented in the Level and Player documentation.

After updating all the objects, it draws them all, depending on of whether the game is paused or not. This is for aesthetic reasons, as it means

```
    //Clear The Last Frame
    c.clearRect(0, 0, 40*u, 20*u);

    //Then Draw everything
    game.levels[currentLevel].draw();
    Player.drawAll();
    HUDElements.drawAll();
}
```

The drawing functions are documented in Level, Player, and HUD, below.

Finally, if the game is in debug mode, it calls the debug function, before calling the render function again.

```
if(game.debug) {
    debug();
}
//Call the next frame
requestAnimationFrame(render);
```

`RequestAnimationFrame();`

Javascript's `window.requestAnimationFrame()` is an iterative function that is used instead of the older `setTimeout()`. Originally, `setTimeout()` was used for animations, as it takes a function and a time period in milliseconds, and calls the function after every time period, so people would have code that looks like this:

```
function render() {
    //Do stuff
}

setTimeout(render,20) //50 times per second
```

This, however is not the most optimised solution, and as such uses more CPU, GPU and RAM than the newer `requestAnimationFrame()`. `setTimeout` also runs regardless of whether or not the game is in focus, and so navigating away from the game still leaves it running, consuming power.

`requestAnimationFrame()` is useful as it runs as soon as the browser is ready, rather than at a fixed time interval, allowing for smoother animations. It also is optimised for the browser it is running in, as "The browser can optimize concurrent animations together into a single reflow and repaint cycle, leading to higher fidelity animation." (Paul Irish, from whom initially obtained the `requestAnimationFrame` shim I am using for my code.)

requestAnimationFrame shim by Paul Irish
https://www.paulirish.com/2011/requestanimationframe-for-smart-animating/

```
/ shim layer with setTimeout fallback
window.requestAnimFrame = (function(){
  return  window.requestAnimationFrame       ||
          window.webkitRequestAnimationFrame ||
          window.mozRequestAnimationFrame    ||
          function( callback ){
            window.setTimeout(callback, 1000 / 60);
          };
})();


// usage:
// instead of setInterval(render, 16) ....

(function animloop(){
  requestAnimFrame(animloop);
  render();
})();
// place the rAF *before* the render() to assure as close to
```

```
// 60fps with the setTimeout fallback.
```

levels.js

Level.js is a core module that creates a class for platformer levels. The module uses some methods of Player.

How to use Level

First create a new level:

```
var level1 = new Level(60,noBg,noFg);
```

Then add objects to it:

```
level1.add(new Platform(16,60,1,4));
```

This should be done a file called `level1.js`, which needs be included in `index.html`

Level Object Constructor

```
constructor(length,background,foreground)
```

The level constructor takes 3 arguments:

`length,` length of the level in tiles. Should be an integer, but any positive real number is valid.

`background` and `foreground`, which are both `Backgrounds`, defined in images.js. `LevelObject.draw()` first draws `background,` then all of the level objects, and then foreground over the entire game. Use these to draw decorations in your game.

Level Object Properties

```
len = length*u;  //converts the length in tiles to length in Game Pixels and
stores it.
offset = 0;  //counter used to store how far the level has scrolled
statics = [];  //statics, dynamics and collectibles are 3 arrays
dynamics = [];  //used to store the 3 different possible types of objects in the
game.
collectibles = []; //read more on these three arrays below.
background = background; //These two properties store the background
foreground = foreground; //and foreground given to the constructor function
scrollLock = false; //this property is used to stop the level from scrolling
when set to true, if you so require.
```

```
index = game.levels.length-1; //used to store the current level number.
```

`static` objects are objects which do not move relative to the co-ordinate system. They are not updated every frame, and neither are they informed when interacted with, unless another module specifically targets them.

`dynamic` objects move between frames, be they enemies, platforms, or projectiles. They are updated every frame, and they are informed when interacted with, so that the objects can interact. All dynamic objects call `Level.colcheck()` to check for collisions.

`collectible` objects are also updated every frame, but they do not collide with anything, and their update function checks for collisions with players and nothing else. This could be changed by modifying `collectibles.js` or writing a custom module, if the user requires it.

Type	Updated every frame?	Checks for collisions?	Informed of collisions
`statics`	No	No	No
`dynamics`	Yes	Yes	Yes
collectibles	Yes	Only with Players by default	No

Level Object Methods:

```
add(...args)
colCheck(obj)
draw()
reset()
scroll(x)
update(multiplier)
```

`LevelObject.add(...args)`

Takes a list of Game Objects as arguments, and if they are of a valid type(static, dynamic, or collectible), adds them to the relevant arrays in the `LevelObject`, if an object is of an invalid type, it outputs an error to the console with the offending object, but continues adding objects to the scene.

It is an example of a variadic function, that is a function which can take a variable number of arguments.

Note: While technically all javascript functions are variadic, `LevelObject.add()` is the only one I have written which will actively handle extra arguments. Passing extra values to any other function will simply cause those arguments to be discarded.

The `LevelObject.add(...args)` function contains the following code:

```
add(...args) {
    for(var i in args) {
      var obj = args[i];
      if(obj.constructor.type == "static") {
        this.statics.push(obj);
      } else if(obj.constructor.type == "dynamic") {
        this.dynamics.push(obj);
      } else if(obj.constructor.type == "collectible") {
        this.collectibles.push(obj);
      } else {
        console.error("Error adding to scene: object", obj,"is not of a
valid type");
      }
    }
  }
```

Looking at this function lets us talk about three key areas:

- The `...args` syntax
- The `this` syntax
- The difference between `console.error(error)` and `throw error`

The ...args syntax

The `add` function needs to be able to take a variable number of arguments, as users may decide to add 1,2,5,10 or even 100 different things to a scene at one time, and it is inefficient to call the `add` method every time they wish to add to a scene. This is solved by the use of the *rest parameter*. The leading ellipsis is called the *rest parameter* in javascript, and tells the interpreter to store all arguments given to the function in the succeeding parameter as an array, in this case, in the parameter `args.` This array can then be iterated through, and its values can be processed. Here, I simply look at the type of each object, and push it to the relevant array within the `LevelObject` the method was called upon. This is done using the `this` syntax.

The this syntax

These methods are all called on an instance of the Level class, for example:

```
level1.add(
    new Platform(0,10,40,10,dirt)
)
```

The `add` method, however needs access to the `level1` object in order to actually add the parameters to their relevant arrays. The `this` keyword returns `level1`, within the scope of `add`, meaning that the `LevelObject` does not need to be passed in as an argument, making the code the user writes easier to read.

```
this.statics.push(obj);
//this returns the level the add method was called on, in this case
//level1
```

Console.error vs throw Error

I go more into depth on errors below, but as this is the only instance of a console.error call, I speak about the differences between the two approaches here.

These are both used to tell the programmer than an action they have attempted is invalid, even if it is programmatically valid. For example, passing "3" as a string into a while loop is technically valid (explained below) but evidently does not make sense. This is where a programmer may wish to check the the type of the parameter, and if it is invalid, they should output an error to the console.

This is normally achieved with the throw keyword, which will output an error to the console and stop running the current script, be it a method, procedure or the entire program.

Example:

```
throw "DependencyError: Module platform requires module box to run";
```

However, there may be occasions where an error needs to be caught, but the code should keep running. I needed this for the add function, in case an invalid object was passed to it. If an invalid object was given, it should be discarded and an error should be output, but the code should keep on trying to add the rest of the items. While traditionally this sort of behaviour is implemented with a `try catch` structure:

```
try {
    //Run some code
} catch(e) { //e contains whatever error the above code throws
    console.log(e) //quietly output the error
}
```

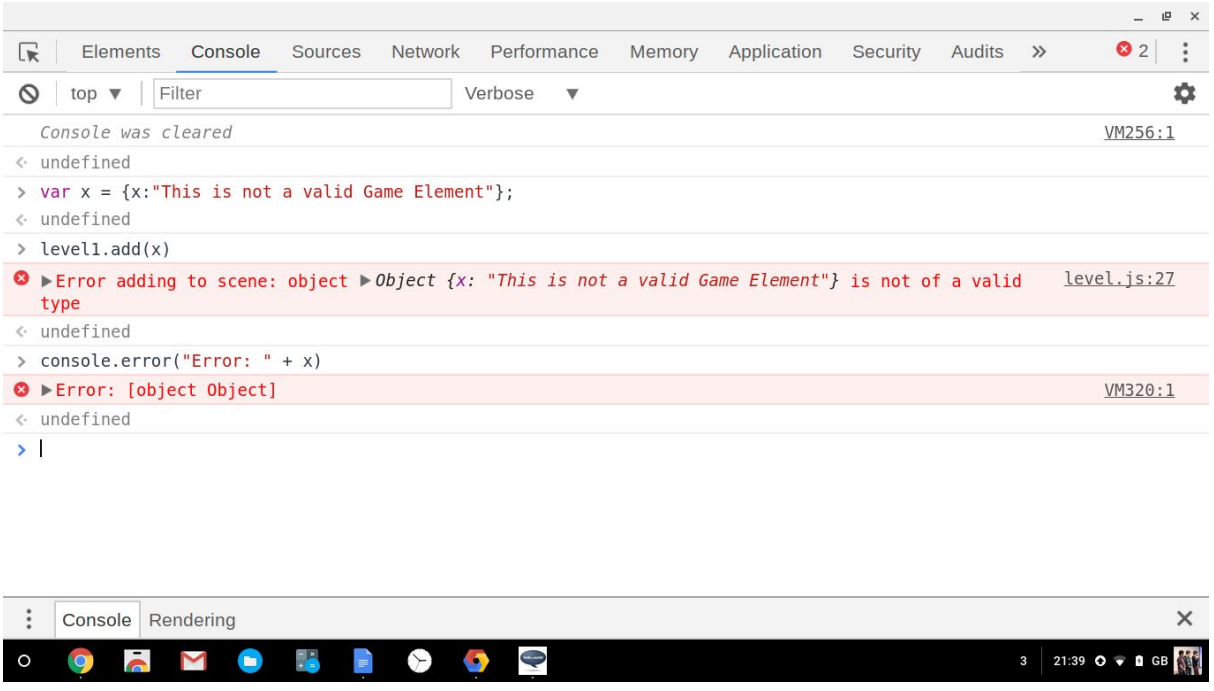
This would not work for the add function, as there is nothing programmatically wrong with attempting to push an invalid object to the level, as arrays can contain any type of data. Running the code with a try catch loop would just cause the object to be added to an array or to be discarded. The user would not what happened to the object and would assume it was added correctly, and unexpected behaviours could occur within their game.

This could be fixed by adding throw statements within the try statement, but this would make the code cluttered as the try catch statement would have to run every time the code loops, that is,once for every object added, of which there may be hundreds.

`console.error` solves this problem. It can be used precisely in the same way as the throw keyword, but it does not cause the interpreter to stop running the add function. It also, coincidentally, is a variadic function, and will output all of the arguments given to it to the console.

```
console.error("Error adding to scene: object", obj ,"is not of a valid type");
```

`obj` is passed as an argument rather than concatenated into the string so that the user can inspect its properties in the console.



`LevelObject.colCheck(obj)`

Takes a Game Object, `obj`, and checks for collisions between any of the `LevelObject.statics`, or any of the `LevelObject.dynamics`. It returns an array of directions (either `"u"`,`"d"`,`"l"` or `"r"`), and after each direction, the name of the type of object it collided with. Any dynamic objects `obj` collides with will have their `collision` methods called with `obj` as an argument.

The actual mathematical collision detection is done by `Box.colCheck(obj1,obj2);` , this method simply loops through the level and checks obj with everything in the stage. I document `Box.colCheck` later,

`LevelObject.draw()`

Calls the `draw()` method of, in order: `LevelObject.background`, `LevelObject.statics`, `LevelObject.dynamics`, `LevelObject.collectibles`, and `LevelObject.foreground`

The code for this function follows this pattern:

```
draw() {
    this.background.draw(this.offset);
    var i = this.statics.length;
    while(i--) {
      this.statics[i].draw();
    }
    //Loops through this.dynamics and this.collectibles the same way.
```

While the method itself is uninteresting, I think this is an appropriate time to discuss the `while(i--)` construct.

```
i = 3
while(i--) {
    //Loops 3 times
}
```

While a more traditional Javascript for loop would be more human readable, i have written my loops in this fashion for speed reasons, which are discussed in this StackOverflow answer: http://stackoverflow.com/a/5349485/1707126. While there are marginally faster methods of looping, they require more variables and checks, meaning more memory is needed to run them. The reason I need to focus on speed is that there are many of these loops, at least ten in the level class alone, all of which loop through possibly hundreds of objects depending on the level, and this is done every frame, of which there are 60 per second. That means I would need to iterate through around 1000 objects in 16.7 milliseconds.

The way this loop works is due to Coercion. The while loop tries to evaluate the expression in the brackets as a boolean value, and it will coerce, or Typecast, the returned variable into a boolean if it is not one already. This also relies on the fact that in Javascript, 0 is a *falsy* value. That is, when you convert the integer 0 to a boolean, it evaluates to false, whereas any other number evaluates to true

The example loop also requires the decrementation operator to function as well, which will return the variable, and then decrease it by one.

These three all combine together so that every time the condition is evaluated, something like this happens

```
Check conditional
“i--”
i returns 3
3 evaluates to true
Decrement i //i = 2
Loop to start
//Once i = 0
Check conditional
“i--”
i returns 0
0 evaluates to false
Decrement i
Continue program
```

It is a difficult concept to grasp if you are new to Javascript, but it is useful as it requires very little code and is very fast compared to the traditional for loop.

`LevelObject.reset()`

Scrolls the level object back to the start, and then loops through `dynamics` and calls each object's `reset()` method.

`LevelObject.scroll(x)`

Loops through `statics`, `dynamics`, and `collectibles` and scrolls them by decreasing each objects x co-ordinate by `x`. Increments `LevelObject.offset` by `x`, before calling `Player.scroll(x);`

```
i = this.collectibles.length;
while(i--) {
      this.collectibles[i].x -= x;
}
```

```
this.offset += x;
Player.scroll(x);
```

The x value of each game object is decreased in order to move them to the left, but the offset is increased as the “camera” is moving in the positive x direction

`LevelObject.update(multiplier)`

This method first calls the `update(multiplier)` methods for all `dynamic` and `collectible` objects in `LevelObject`, before running code to check if the level should be scrolled, and handling level changing if the player is off screen. The code for this method is more or less self explanatory, and is shown below

```
update(multiplier) {
    //Update all the entities that may have changed
    var i = this.dynamics.length;
    while(i--) {
      this.dynamics[i].update(multiplier);
    }

    i = this.collectibles.length;
    while(i--) {
      this.collectibles[i].update();
    }
    //Get the rightmost player
    var lead = Player.getLeader();

    //If they are: moving right, and in the center of the screen and not
at the end of the level, and the level is not scrollLocked, scroll the
level left
    if(
        (lead.vX > 0.1) && (lead.x > 18*u) &&
        (this.offset < (this.len - (tilesX*u))) && (!this.scrollLock)
    ) {
      this.scroll(lead.vX);
    }
    //If they are moving left, and in the center of the screen and not
at the start of the level, and the level is not scrollLocked, scroll the
level right
    if(
        (lead.vX < -0.1) && (lead.x < 17*u) &&
        (this.offset > 0) && (!this.scrollLock)
    ) {
        this.scroll(lead.vX);
```

```
    }

    //Make sure the level is not scrolled past its max and minimum
    if(this.offset < 0) {
        this.scroll(-this.offset);
    }
    if(this.offset > this.len - tilesX*u) {
        this.scroll((this.len - (tilesX*u)) - this.offset);
    }

    //Handle level switching
    if(lead.x > tilesX*u) {
        Player.moveAll(1,10);
        currentLevel++;
    }
    if((lead.x < 0) && (currentLevel > 0)) {
        Player.moveAll((tilesX-1)*u,10);
        currentLevel--;
    }
    if(currentLevel >= game.levels.length) {
        currentLevel = 0;
    }
}
```

box.js

box.js is the most basic module, from which all of the other modules inherit properties. It implements a rectangle and a collision detection algorithm, and is used to check collisions between objects.

```
//Push "box" to the list of modules included
modules.push("box");

//Define the box Class
var Box = class Box {
    constructor(x, y, width, height, solid) {
        this.x = x*u;
        this.y = y*u;
        this.h = height*u;
        this.w = width*u;
        this.s = solid;
    }

    draw() {
        c.fillStyle = "green";
```

```
        c.fillRect(this.x,this.y,this.w,this.h);
    } //code continues
```

The class is not intended to be added to the scene, and as such, it is missing a `Tile` property for drawing. Instead, modules that build on it govern how it is drawn. The `x` and `y` properties are used to position the box within the level, and the `w` and `h` properties set the width and height of the box, respectively. The solid property is used to control how the box behaves when collided with.

```
//Collision algorithm from somethinghitme.com
static colCheck(a,b) {
  // get the vectors to check against
  var vX = (a.x + (a.w/ 2)) - (b.x + (b.w / 2)),
      vY = (a.y + (a.h / 2)) - (b.y + (b.h / 2)),
      // add the half widths and half heights of the objects
      hWidths = (a.w / 2) + (b.w / 2),
      hHeights = (a.h / 2) + (b.h / 2),
      colDir = null;
// if the x and y vector are less than the half width or half height,
they we must be inside the object, causing a collision
      if (Math.abs(vX) < hWidths && Math.abs(vY) < hHeights) {
      // figures out on which side we are colliding (top, bottom, left,
or right)
          var oX = hWidths - Math.abs(vX),
              oY = hHeights - Math.abs(vY);
          if (oX >= oY) {
              if (vY > 0) {
                  colDir = "t";
          //If the second item is solid, move the first outside of it.
                  if (b.s) {
                      a.y += oY;
                      a.vY = 0;
                  }
              } else {
                  colDir = "b";
                  if (b.s) {
                      a.y -= oY;
                      a.vY = 0;
                  }
              }
          } else {
              if (vX > 0) {
                  colDir = "l";
                  if (b.s) {
```

```
                    a.x += oX;
                    a.vX = 0;
                }
            } else {
                colDir = "r";
                if (b.s) {
                    a.x -= oX;
                    a.vX = 0;
                }
            }
        }
    }
    return colDir;
  }
};
```

I modified this collision detection algorithm from http://www.somethinghitme.com/2013/04/16/creating-a-canvas-platformer-tutorial-part-tw/ I added the conditionals to only move the object out of the other if the object was solid. I use it to check for collisions between Game Objects. I used this algorithm rather than writing my own as I needed a fast, concise algorithm as this function would be run many times per frame, perhaps hundreds, and all of the condition-based algorithms I designed were too slow and inefficient for the game to run properly.

`Box.colCheck(a,b);` is a *static* method. Static methods are methods of a class, rather than instances of a class. I decided to make this a static method as collisions may be checked between objects that do not inherit the Box Class, and making the method static makes it globally accessible.

projectiles.js

Projectiles are boxes that can move. The code for this class below and I will be analysing it in detail as there are multiple constructs that are repeated across many modules.

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module projectile";
} else {
  //Push "projectile" to list of included modules
  modules.push("projectile");

  //super() calls the Box constructor, before adding vX and vY
properties, which store XY velocity
  var Projectile = class Projectile extends Box {
    constructor(x,y,height,width,vX,vY,solid) {
```

```
      super(x,y,height,width,solid);
      this.vX = vX;
      this.vY = vY;
    }

    move(multiplier) {
      this.x += this.vX * multiplier;
      this.y += this.vY * multiplier;
    }
  };
  Projectile.type = "dynamic";
}
console.log("Projectile.js Loaded");
```

The aspects of this module I want to talk about, before I discuss how the module works in detail, are:

- Module dependency management
- Inheritance
- Multipliers
- Class properties

Module Dependency Management

In `main.js`, a global variable named `modules` is defined, which is an array of strings which I use to check for dependencies. Every time a module is loaded, it pushes its name to the `modules` array. This array is then searched by other modules when they are loaded to ensure all the modules they depend on have been loaded.

For example, the `box` module will push "`box`" to `modules`. Then, when `projectiles` begins loading, it will check modules for the "`box`" string, and if it is returned, it will continue loading. It checks for the presence of these strings using the `Array.indexOf(x)` method, which will return the position of `x` within `Array`, or `-1` if it is not found. The `throw` keyword, which I discussed within `level.js` is used to make sure that the code terminates and alerts the user that they are missing a module, else their code would silently fail and they would have to spend time looking for the error.

Inheritance

The projectile module inherits all of the properties of the `box` class, and this is achieved through the use of the `extends` keyword, along with the `super()` function. The use of inheritance means that I do not have to define the properties over and over, I can simply pass them to the parent to deal with them, meaning that within `projectile.js`, I can write code that focuses on what makes projectiles different to boxes. The extends keyword is used to indicate which class the current class inherits properties from, and the `super()` function generates an object of this parent class and returns it so that the

child constructor can continue modifying it.

Example:

```
//Calling the Projectile Constructor:
new Projectile(10,10,1,1,5,0,true);

//Actually first calls the Box constructor, and returns a box of the
specified dimensions,
constructor(x,y,height,width,vX,vY,solid) {
    super(x,y,height,width,solid);
    this.vX = vX;
    this.vY = vY;
//Before adding properties unique to the Projectile class to the object
}
```

Multipliers

The multiplier value from `game.js` is used to move the projectiles depending on the speed of the game and the value of `multiplier`.

```
this.x += this.vX * multiplier;
```

If the game is running at 30fps instead of 60, the value of multiplier will be 2, and the projectile will move twice as far in that frame as it normally would.

Class Properties

ES6 does not actually implement classes as other languages have, and as such, classes can be given properties much like regular objects can, and I use this to denote the type of each module. This is used by `LevelObject.add()` in order to make sure only valid objects are added to the scene.

```
Projectile.type = "dynamic";
```

This property is read by the add method to make sure that all Projectiles are added to the dynamic array.

```
//From Level.add()
if(obj.constructor.type == "dynamic") {
        this.dynamics.push(obj);
}
```

`obj` being the object being added, `obj.constructor` being the object's class, and `obj.constructor.type` returning the `type` property we just set.

The move() function

This function simply moves the projectile across the screen depending on its x and y velocity, and depending on the frame rate of the game. This process is called **iterative integration**, as I am iteratively integrating the player's velocity to give the player's new position. Acceleration is achieved by changing the player's velocity between frames.

player.js

Player.js is a core module that implements a platformer Player class. This class depends on the Projectile and Input modules.

Players are added to the Game Object every time they are instantiated by their constructor:

```
new Player(1,1,{left:65,right:68,up:32},Sprite);
```

This will spawn in a `Player` at `(1,1)`, with the controls `A`, `D` and `Space` for Left, Right and Jump respectively. The numbers in the third argument correspond to keyboard keycodes. `Sprite` is a `PlayerSprite` defined in `sprites.js`. I document inputs and images later.

Player Object Constructor

```
constructor(x,y,controls,sprite)
```

The player constructor takes in four arguments: `x`, `y`, `controls`, `sprite`.

- `x` and `y` are coordinates, in `u`, where the player is initially spawned. `u` is defined in `main.js` as the number of pixels per tile.
- `controls` is an object which is used to assign keys to the player. By default, it should have the properties left, right and up, though other modules may require users to add extra keys to the `controls` object, for example, `switches.js` requires an `open` key to be added in order to trigger the switch.
- `sprite` is used to draw the player. It can be any valid Sprite, but not a Tile. I discuss images later, but briefly, it is an `ImageObject` that is drawn at the player's location every frame.

Player Object Properties

Properties unique to the Player class are:

```
controls //stores the controls object passed to the Player constructor.
vXmax = (1/6)*u //Constant value, limits players x velocity.
vYmax = (1/4)*u //Constant value, limits players y velocity.
friction //used to store coefficient of friction on current surface.
         //Frictional values defined in main.js
```

```
isJumping //stores whether the player is currently jumping or not,
          //used to prevent jumping when in the air
sprite //stores the sprite objects passed to the constructor
health //contains the current health of the player as a percentage
playerNumber //Constant, equal to player's position in game.players + 1.
```

Player Object Methods

```
checkDir(dir)
checkKeys(multiplier)
draw()
hit(damage)
kill()
update(multiplier)
```

`PlayerObject.checkDir(dir)`

Check the direction of every collision, and the type of object the Player has collided with, and changes the player's values depending on the values within `dir`. `dir` is an array of values returned by `LevelObject.colCheck(obj).`

`PlayerObject.checkKeys(multiplier)`

Checks which keys are pressed and moves the player depending on the value of `multiplier`. If the left/right key is pressed, and the players x velocity is less than the maximum velocity, it will increase the player's x velocity.

If the `up` key is pressed, and the player is not jumping, the players y velocity is decreased (increased in the up direction as (0,0) is the top left corner) and `PlayerObject.isJumping` is set to true.

The function then applies `friction` and `gravity` to `PlayerObject.vX` and `vY` respectively, in order to allow `PlayerObject.move()` to move the player appropriately.

`PlayerObject.draw()`

Runs `PlayerObject.sprite.draw()` and passes it the current state of `PlayerObject`. See Images for more.

`PlayerObject.hit(damage)`

Decreased `PlayerObject.health` by `damage`, and if `health` falls below `0`, calls `PlayerObject.kill();`

`PlayerObject.kill()`

Called when the player falls offscreen or `health` falls below zero. First calls `reset()` on the current `LevelObject`, before resetting `PlayerObject`'s `x`, `y`, `w`, `h`, `vX`, `vY`, and `health` to the default values. If users wished to change these values they need to modify the values within this function.

`PlayerObject.update(multiplier)`

This is the master function used to update the player. It follows this algorithm:

1. Check if the player has fallen below the screen and kill the player if it has.
2. Check which keys have been pressed using `PlayerObject.checkKeys()`
3. Move the player using the `Projectile` classes `.move()` method
4. Fetch a list of what the player has collided with, using the `Level` classes `.colCheck(obj)` method
5. Modify the player depending on what it has collided with, using `PlayerObject.checkDir();`

Code:

```
//Polymorphic routine, called on every dynamic object every frame
//Checks keys pressed, moves and checks collisions
//with all items in the current level
//Dir contains a direction, or null if there is no collision.
update(multiplier) {
    if(this.y > tilesY*u) {
        this.kill();
        return;
    }
    this.checkKeys(multiplier);
    this.move(multiplier);
    var dir = game.levels[currentLevel].colCheck(this);
    this.checkDir(dir);
}
```

Update is a *polymorphic routine*, that means more simply, all dynamic objects have an `update()` method, but each runs different code. This is important it means I can simply call `update()` on all game elements, and the specifics on how to update them is handled within the class.

Player Class Static Methods:

```
static drawAll()
static getLeader()
static moveAll(x,y)
```

```
static scroll(x)
static updateAll(multiplier)
```

`Player.drawAll();`
Loops through all players and calls their draw function.

`Player.getLeader();`
Loops through all players and returns the rightmost player. It does this by comparing `x` coordinates and the `PlayerObject` with the highest `x` coordinate is returned.

`Player.moveAll(x,y);`
Loops through all players, and sets all their x coordinates to `x*u` and y coordinates to `y*u`

`Player.scroll(x);`
Loops through all players and decreases their x coordinate by `x`, moving them to the left by `x` game pixels.

`Player.updateAll(multiplier);`

Loops through all players and calls their `update()` method, passing in `multiplier` as an argument.

input.js
Input.js handles how mouse and keyboard events are handled by the game:

```
modules.push("input");

window.addEventListener("load",function() {
  canvas.addEventListener("mousedown", CanvasClickHandler, false);
  canvas.style.zIndex = 1;
});

function CanvasClickHandler(e) {

}

function getPosition(event) {
  evX = event.x;
  evY = event.y;
  evX -= canvas.offsetLeft;
  evY -= canvas.offsetTop;
  return {
```

```
    canvasX: evX,
    canvasY: evY,
    screenX: event.x,
    screenY: event.y,
    gameX: tilesX * u * evX/window.innerWidth,
    gameY: tilesY * u * evY/window.innerHeight
  };
}

document.body.addEventListener("keydown", function(e) {
  if ((!keys[27]) && (e.keyCode == 27)) {
    if(!game.paused) {
      game.pause();
      pause.style.display = "block";
    } else {
      game.play();
      pause.style.display = "none";
    }
  }
  keys[e.keyCode] = true;
});
document.body.addEventListener("keyup", function(e) {
  keys[e.keyCode] = false;
});

console.log("Input.js Loaded");
```

Fetching Keyboard Events

When a key is pressed, the global keys array is updated with a true value at `keys[keycode]`

Keycodes are numbers assigned to keys and you can find these using keycode.info
For example, the `D` key is keycode `68`. When `D` is pressed, `keys[68]` returns `true`.
If the key has never been pressed, `undefined` is returned, else `false`, if the key is not currently being pressed.

For example, the `keys` array will most likely look something like this:

```
  [undefined × 32, false, undefined × 35, true]
```

This shows that `keys[32]` (space) was pressed at some point in time, and `keys[68]` (d) is currently being pressed. Keypresses are usually checked with code similar to this

```
if(keys[obj.controls.right]) {
```

```
    //do stuff
}
```

Where `obj.controls.right` is an integer corresponding to a keycode. This code is another example where Javascript's coercion, or typecasting, is useful, as the `if()` statement automatically converts the `undefined` value to `false`, so that we do not need to fill the array with `false`s, and neither do we need to check if the value is `undefined`, it is all handled by the compiler.

The boolean values for each key are set using a keydown/keyup event listener:

```
document.body.addEventListener("keyup", function(e) {
  keys[e.keyCode] = false;
});
```

When one of these events occur, they are added to the event queue, and then handled by the interpreter once the render() function terminates. This gives nicer, less obtrusive code, but it does mean input lag of up to 1 frame is possible, though under normal operation, this should not be noticeable at it would be 16/17 milliseconds between frames.

The keydown event listener also has a pause function hardcoded into it using the escape key, meaning that in the event of a game crashing, or if the player simply wishes to leave the game, they can press this key to leave. While I considered adding a "game.pauseButton" property, I decided against this as it could cause confusion with players, as different games would inevitably choose different pause buttons. This module, of course, can still be changed by users, but this means that novice programmers are less likely to change this setting.

```
document.body.addEventListener("keydown", function(e) {
  if ((!keys[27]) && (e.keyCode == 27)) {
    if(!game.paused) {
      game.pause();
      pause.style.display = "block";
    } else {
      game.play();
      pause.style.display = "none";
    }
  }
  keys[e.keyCode] = true;
});
```

`Pause` is an `ID` to a `HTML Element`, which is displayed when the game is paused. Javascript fetches a list of element IDs and creates global variables for each so that they can be accessed without the `document.getElementById()` method.

Fetching MouseDown Events

The input modules adds a method, `getPosition(e)` which returns an object of `[x,y]` values when the canvas is clicked. The `CanvasClickHandler()` function is called every click, and is used to define how the click event should be managed

`getPosition(e)` returns an object with the following properties: (comments wrap to two lines)

```
screenX // Location where pressed on the screen, x
screenY // and y. Depends on client display
canvasX // Location on canvas where pressed on the canvas. Resolution
        // equal to screenX and Y, but offset
canvasY // Depending on client display.
gameX   // Location of click in game pixels
gameY   // Independent of client screen
```

Example Use:

```
function CanvasClickHandler(e) {
    var coords = getPosition(e); // e contains click event
    game.players[0].x = coords.gameX; // Moves player 1 to the location
    game.players[0].y = coords.gameY; // of the click in game.
}
```

images.js

`images.js` implements `Tiles` and `Sprites`, collectively referred to as `ImageObjects` for the game to use to render objects.

ImageObject types

There are 7 image types of image objects

```
Background
BlockColTile
BlockColSprite
RepeatingTile
Sprite
SpriteSet
Tile
```

These can be divided into 2 categories: Tiles and Sprites.

Tiles are drawn at custom positions with varying sizes.
Sprites are usually drawn at a fixed position in the level

`Tile`

Tiles take a filepath, and an cx, cy, cw, ch, used to crop the image for the tile. Their draw() method takes an x, y, w, h to draw the image. The draw() method stretches the image to the correct size if necessary. Both Tiles and Sprites use the Canvas 2D context drawImage method:

```
draw(x,y,w,h) {
    c.drawImage(this.img,this.cx,this.cy,this.cw,this.ch,x,y,w,h);
}

//this.img is a HTML Image Object defined in the constructor function

constructor(url,cx,cy,cw,ch) {
    this.img = new Image();
    this.img.src = url; //Loads the image from url into this.image
```

`Sprite`

Sprites take a filepath, `cx`, `cy`, `cw`, `ch`, `x`, `y`, `w` `h` and `offsetFactor` The `c` values are used to crop from a source image, the `x`, `y`, `w`, and `h` parameters define where in the level the sprite will be drawn, and how big. `offsetFactor` determines the rate at which the sprite scrolls, in case the user wishes to implement parallax. An `offsetValue` of 0 means the sprite does not scroll with the level, and a factor of 1 means the sprite and level scroll the same amount. Any real value is valid, but values between 0 and 1 are typical.

`BlockColSprite`

Creates a sprite of a solid colour. Accepts any valid CSS colour as a string. `BlockColSprite.draw()` functions the same way as `Sprite.draw()`

`BlockColTile`

Creates a block colour tile. Accepts any valid `CSS` colour as a string, functions the same as a regular tile when drawing. Both `BlockColSprite` and TIle use the Canvas 2D Context `drawRect()` function

```
draw(x,y,w,h) {
    c.fillStyle = this.col;
    c.fillRect(x,y,w,h);
}
```

`RepeatingTiles`

`RepeatingTiles` extend the base `Tile` class by drawing a repeating pattern. This may be useful for large objects. The constructor method takes a filepath to the image you wish to draw. The image cannot be cropped or resized for speed reasons. Repeating Tiles are slow to draw, so where possible, users should avoid them in favour of `Background` or `Tile` images. `RepeatingTiles` are drawn in the same way as regular `Tiles`

```
draw(x,y,w,h) {
    c.translate(x,y);
    var style = c.createPattern(this.img,"repeat");
    c.fillStyle = style;
    c.fillRect(0,0,w,h);
    c.translate(-x,-y);
}
```

The repeating tile method moves the `canvas's` origin to the top left corner of the object to draw, before creating a repeating pattern, drawing it from the (new) origin to the required dimensions, before moving the origin back to the top left corner of the canvas. The `c.createPattern` is the slowest step of this algorithm, and with multiple repeating tiles easily takes 50-60% of the `render` loop's time.

`SpriteSet`

Returns an objects with property layers, which contains an array of `SpriteObjects` provided to the constructor function

`Background`

These are `SpriteSets` with a `draw` method. These are passed to `LevelObjects` to draw behind and infront of the level.

How to use images

Most game elements will take in a Sprite or Tile as an argument, and every frame, the objects `.draw()` method will call the `ImageObjects draw()` method. Due to this, users should define these objects in `sprites.js` as variables so that they can reuse the same `ImageObject` multiple times. For example:

```
//In sprites.js
var dirt = new RepeatingTile("assets/dirt.png");
```

This way, if they want to have more than one `Platform` with the `dirt` texture, you can use the `dirt` variable twice:

```
//In level1.js
level1.add(
    new Platform(0,18,30,2,dirt),
    new Platform(30,17,30,3,dirt)
)
```

As you can see, we did not need to pass in new `RepeatingTile()` twice. Aside from being nicer to read and write, this is also crucial to the running the game, as users will inevitably have hundreds of `platforms` once they start writing their game, and calling the new `RepeatingTile()` method that many times will effect performance.

Backgrounds

Backgrounds are created in a similar way. Users may either define background layers as sprites, and pass them to an array, or they can simply create them all within an array, then pass the array to the `Background()` constructor as so:

```
//In sprites.js
var level1Bg = new Background([
    new Sprite("assets/sky2.png",0,0,100,20,0,0,10000,2000,0.2),
    new Sprite("assets/sky1.png",0,0,40,20,0,0,4000,2000,0)
]);
```

Note: The topmost sprite is rendered last. This is due to the fact that the while-based for loop I mentioned earlier loops from the end of the array to the start.

Note: Backgrounds only except `Sprite` or `BlockColSprite` objects. Attempting to draw `Tile` elements will fail as no parameters are passed to the drawing function.

These backgrounds are passed to the Level constructor when creating new levels. They will be drawn both behind and in front of the level. To add `level1Bg` to `level1`, it needs to be passed to the level `constructor`, like so:

```
//Top of level1.js:
var level1 = new Level(60,level1Bg,noFg);
```

As you can see, I passed "`noFg`" to the constructor as I did not want a foreground on this level. There are three of these "`Invisible Sprites`" which I document below:

Invisible Sprites

These sprites do not render anything. they are called `noBg`, `noFg`, and `noTile`, all of which link back to one object:

```
{draw: function() {}}
```

All arguments passed to it are ignored, and no canvas operations occur. It simply terminates as soon as it is run.

PlayerSprites

A PlayerSprite ImageObject takes a filepath and a data object in its constructor, and these are used to render the Player depending on its current state, The data object has many properties, which are:

```
sw: Sprite Width
sh: Sprite Height
drx: Falling, Facing Right,X Co-ord
dry: Falling, Facing Right,Y Co-ord
dlx: Falling, Facing Left,X Co-ord
dly: Falling, Facing Left,Y Co-ord
urx: Jumping, Facing Right,X Co-ord
ury: Jumping, Facing Right,Y Co-ord
ulx: Jumping, Facing Left,X Co-ord
uly: Jumping, Facing Left,Y Co-ord
lbx: Skidding, facing left, X Co-ord
lby: Skidding, facing left, Y Co-ord
rbx: Skidding, facing right, X Co-ord
rby: Skidding, facing right, Y Co-ord
l1x: Walking Left, Frame 1, X Co-ord
l1y: Walking Left, Frame 1, Y Co-ord
l2x: Walking Left, Frame 2, X Co-ord
l2y: Walking Left, Frame 2, Y Co-ord
r1x: Walking Right, Frame 1, X Co-ord
r1y: Walking Right, Frame 1, Y Co-ord
r2x: Walking Right, Frame 2, X Co-ord
r2y: Walking Right, Frame 2, Y Co-ord
slx: Standing, Facing Left, X Co-ord
sly: Standing, Facing Left, Y Co-ord
srx: Standing, Facing Right, X Co-ord
sry: Standing, Facing Right, Y Co-ord
frn: Frame Number
afn: Animation Frame Number
```

These can be accessed in the following way: `PlayerSpriteObject.d.PropertyName` if you need to. If you simply need to hookup your spritesheet to the existing animation engine, simply get the x and y co-ordinates of the relevant frames, and assign them to the above properties. The `PlayerSpriteObject.draw()` method will handle the animation.

Modifying the Animation Function If users wish to add more frames for the animation, they need to modify the `PlayerSpriteObject.draw()` function. This will require a thorough understanding of Javascript and CAGE, and as such is restricted to the more advanced users.

The process for adding animation frames involves:

- Add the X and Y coordinates of the frames you want to add to the data object
- Add a conditional for when you want the frame to display to the `PlayerSpriteObject.draw()` function. For example:

```
if(player.x == 300) { //When the player being drawn is at x = 300
    //Draw the player. lx and ly should be replaced with the properties
    //you added to the `PlayerSprite` object.
c.drawImage(this.img,this.d.lx,this.d.ly,this.d.sw,this.d.sh,player.x,pl
ayer.y,player.w,player.h);
    return //This line is required, else the engine will draw over the
sprite
}
```

Mob Sprites

MobSprites are not passed to mobs, which is different to all other images. Instead they are passed to a static `MobSprite.draw()` method, which cannot be modified yet and handles drawing mobs. To skin their own mobs, users need to match the format of the sprite file, as this method is not currently flexible.

Collectibles

Collectibles implement objects which can be picked up by the player

```
var Collectible = class Collectible extends Box {
    constructor(x,y,w,h,tile) {
        super(x,y,w,h,false);
        this.collected = false;
        this.t = tile;
    }

    update() {
        var i = players.length;
        var dir = "";
        while(i--) {
            dir = Box.colCheck(this, players[i]);
            if(dir !== null) {
                this.collect(players[i]);
            }
```

```
        }
    }

    collect() {
        this.collected = true;
    }

    draw() {
        if (!this.collected) {
            this.t.draw(this.x,this.y,this.w,this.h);
        }
    }
};
```

The collectible class on its own only implements a non-solid block which disappears when a player touches it, but other modules can redefine the `collect` function in order to add custom behaviours, as can be seen in the `Coin` and `sizePowerUp` modules.

Ancillary Modules

The modules I have mentioned thus far are all core modules, as I said in my design documentation. I have written the following ancillary modules to add functionality to user's platformer games:

- AI
- Bouncy Platforms
- Breakable Platforms
- Coins
- Crates
- Doors
- Icy Platforms
- Mobs
- Moving Platforms
- Platforms
- Switches
- Size Power Ups

AI

The Ai module adds movement to objects, namely Mobs and Moving Platforms. There are 5 types of simple AI:

- `NoAI`
- `StaticAI`
- `LinearAI`
- `PatrolAI`
- `VerticalPatrolAI`

`NoAI()`

```
function NoAI() {
  return;
}
```

The `NoAI` function simply returns as soon as it invokes, leaving the object fully under the influence of its own momentum and gravity.

`StaticAI()`

```
function Static() {
  this.vX = 0;
  this.vY = 0;
}
```

The `StaticAI` function is used to fix an object in position relative to the level, in order to create floating entities

`LinearAI()`

```
function LinearAI() {
  this.vX = this.aiData.vX;
}
```

The `LinearAI` function is used to move an object at a fixed horizontal velocity, and is the simplest of the non-trivial AIs. The `aiData` object is defined and given to an object when it is instantiated, and is accessed and modified by the different `ai` functions when needed. For the `LinearAI`, users need to define the horizontal velocity `vX` the object should move at, and the object `vX` is set to this `aiData vX` every frame, ready for the `Projectile.move()` method.

`PatrolAI()`

```
function PatrolAI() {
    if(this.aiData.dir == 1) {
        this.vX = this.aiData.vX;
    }
    if(this.aiData.dir == -1) {
        this.vX = -this.aiData.vX;
    }
    var offset = game.levels[currentLevel].offset;
    if(this.x+offset <= this.aiData.x1*u) {
        this.aiData.dir = 1;
    }
    if(this.x+offset >= (this.aiData.x2*u) - this.w) {
        this.aiData.dir = -1;
    }
}
```

The `PatrolAI` function is used to move an object between two fixed points, `x1` and `x2` at a fixed velocity `vX`. The `aiData` object needs the `x1`, `x2`, `vX` and `dir` properties defined when it is instantiated. `dir` is either -1 or 1 and is used to define which direction the object travels in. The `x1` and `x2` properties are given in `u` and are not strict boundaries. The object may travel past these points by up to 1 `vX`'s worth of pixels. However, the object will always reverse directions if it is past these two markers.

`game.levels[currentLevel].offset` is stored in a local variable in order to prevent the interpreter having to access the property multiple times.

`VerticalPatrolAI()`

```
function VerticalPatrolAI() {
  this.vY = this.aiData.dir * this.aiData.vY;
  this.vY -= gravity - 1;

  if(this.y <= this.aiData.y1*u) {
    this.aiData.dir = 1;
  }
  if(this.y >= (this.aiData.y2*u)) {
    this.aiData.dir = -1;
  }
}
```

Functionally, the `VerticalPatrolAI` function is identical to the `PatrolAI` function, but it moves vertically, however it allows us to discuss some of the finer details of how ingrained dependencies are within this system.

The `VerticalPatrolAI` is designed to work with levels that do not scroll vertically, and as such would likely fail if a different type of level module is used, say one which scrolled in both the `x` and the `y` directions. This is not a direct dependency, as no code is shared between the `Level` and `AI`. Instead, the `AI` module assumes the AI functions are being called in a logical context. I could, if I wished, generalised the `Level` module to add a `Y` height and `Y` offset, but I wished to create a vertically locked platformer, akin to the original *Super Mario Bros.*

To stop users mixing modules that should not be used together, I could add simply add a conditional checking for dependencies, or I could warn the user in the documentation for the module, however as I currently only have one type of level module, I decided against this. Instead, in future, I would update this module to be level-implementation independent, so that the `ai` would work regardless of game type.

Bouncy Platforms

```
var BouncingPlatform = class BouncingPlatform extends Box {
    constructor(x,y,w,h,tile) {
        super(x,y,w,h,false);
        this.t = tile;
        this.vX = 0;
        this.vY = 0;
    }

    draw() {
        this.t.draw(this.x,this.y,this.w,this.h);
    }

    collision(obj,dir) {
        if(dir == "b") {
            obj.vY = -0.9*Math.abs(obj.vY);
            if(obj.vY < -obj.vYmax) {
                obj.vY = -obj.vYmax;
            }
            if(Math.abs(obj.vY) < 1) {
                obj.vY = 0;
            }
            obj.vY -= gravity;
        }
    }
```

```
    update() {}
    reset() {}
};
```

The bouncing platforms are simply non-solid platforms, with a collision function that updates the colliding objects y velocity. It first reflects the object's `vY`, before decreasing it by 10%. Then, if the objects vY is greater than its maximum `vY`, its vY is set to the maximum. If the objects `vY` is very small, it is set to zero to allow the object to come to rest when the bounce height is low.

Breakable Platforms

```
constructor(x,y,w,h,tile,data) {
    super(x,y,w,h,tile);
    this.iy = y*u;
    this.d = data || {};
}

reset() {
    this.destroyed = false;
    this.s = true;
    this.y = this.iy;
}

collision(obj,dir) {
    if(obj.constructor.name == "Player") {
        if(dir == "t") {
          this.destroyed = true;
          this.s = false;
          this.y = tilesY * u;
          if(this.d.item) {
            game.levels[currentLevel].add(this.d.item);
          }
        }
    }
}
```

Breakable platforms are the same as normal platforms, but when touched by the top of a player, it is destroyed, becomes non-solid and moves offscreen. If the object has a data object with an item specified, that object is added to the current level.

The logical OR operator is used to check if the property is defined. This uses the same type coercion I mentioned earlier.

```
var x = a || b;
```

a is evaluated first, if it is not a falsy value, its value is assigned to x, else b is assigned, regardless of its value

Coins

```
constructor(x,y,t) {
    super(x,y,0.3,0.3,t);
}

collect(x) {
    if(!this.collected) {
        var collectedBy = x;
        if(!collectedBy.coins) {
            collectedBy.coins = 0;
        }
        collectedBy.coins++;
        this.collected = true;
    }
}
```

The coin modules add collectible items which increase an object's `coins` count. It sets this value to zero if it is not already set.

Crates

```
collision(obj,dir) {
    if(!this.isOpened) {
        if(obj.constructor.name == "Player") {
            if(keys[obj.controls.open]) {
                this.destroy();
            }
            if ((dir == "l") || (dir == "r")){
                this.vX = obj.vX;
            } else if(dir == "b") {
                obj.y = this.y - obj.h;
                obj.vY = -gravity;
            }
        }
    }
}

destroy() {
    if(!this.isOpened) {
        if(this.c) {
            this.c.x = this.x;
            this.c.y = this.y;
```

```
            game.levels[currentLevel].collectibles.push(this.c);
        }
        this.isOpened = true;
    }
}
```

Crate are pushable boxes, which can contain items. The items are released when the player presses an `open` button when touching the crate.

Doors

Doors are toggleable platforms, with an `open`, `close`, and `update()` method.

Icy Platforms

Icy platforms are platforms which have a low frictional value. The friction is handled by the Player class.

Mobs

```
collision(obj,dir) {
    //Dynamic Object has collided with mob
    var objType = obj.constructor.name;
    if(objType == "Player") {
        if(dir == "b") {
            this.aiHit();
            obj.vY = -1*obj.vYmax;
            obj.isJumping = false;
        } else {
            obj.hit(this.aiData.dmg);
            if(dir == "l") {
                obj.vY = -obj.vYmax;
                obj.vX = obj.vXmax;
                obj.isJumping = true;
            }
            if(dir == "r") {
                obj.vY = -obj.vYmax;
                obj.vX = -obj.vXmax;
                obj.isJumping = true;
            }
        }
    }
    if(objType == "Mob") {
        if(dir == "r") {
            this.x -= obj.w;
            obj.x += this.w ;
        }
        if(dir == "l") {
            this.x += obj.w;
```

```
            obj.x -= this.w ;
        }
    }
}
```

Mobs are moving enemies. They are projectiles with added methods. They move using the AI functions i described earlier every frame.

When they collide with the left or right side of a player, they call the players “hit” method, before throwing the player away from the mob. If the mob touches the bottom of a player, the mobs hit handling method, aiHit is called, and the player bounces off the mob.

If two mobs collide, they switch places to allow them to pass through each other.

Moving Platforms

Moving Platforms are platforms which move every frame using the behaviours defined in the AI module.

Platforms

Platforms are solid boxes which the player can walk on. They are the simplest type of ancillary module.

Switches

```
constructor(x,y,sprite1,sprite2,defaultPosition,entity) {
    super(x,y,1,2,false);
    this.on = defaultPosition;
    this.defaultPosition = defaultPosition;
    this.entity = entity;
    this.s1 = sprite1;
    this.s2 = sprite2;
}
```

These are non-solid boxes, which, when collided with by a `player` with their `open` key pressed, fires the `update` method of their `entity` (usually a `door`). And `switches` position from `on` to `off`, or vice versa. `s1` and `s2` are two different sprites drawn depending on the state of the switch.

Size Power Ups

```
collect(x) {
      if(!this.collected) {
        this.x = this.ix + game.levels[currentLevel].offset;
        this.y = this.iy;

        var collectedBy = x;
        if(!this.permanent) {
```

```
          this.collected = true;
        }
        collectedBy.w = this.factor * u;
        collectedBy.h = this.factor * 2 * u;
      }
    }
```

The size powerup sets the size of the player to factor times the normal size of the player. The powerup can be constructed to be permanent or collectible using a single boolean argument. The location of the powerup must be reset as the collision detection algorithm does not permit intersecting solid object to occupy the same space. (though if the collision detection algorithm is not checked between two objects they can occupy the same space)

Testing

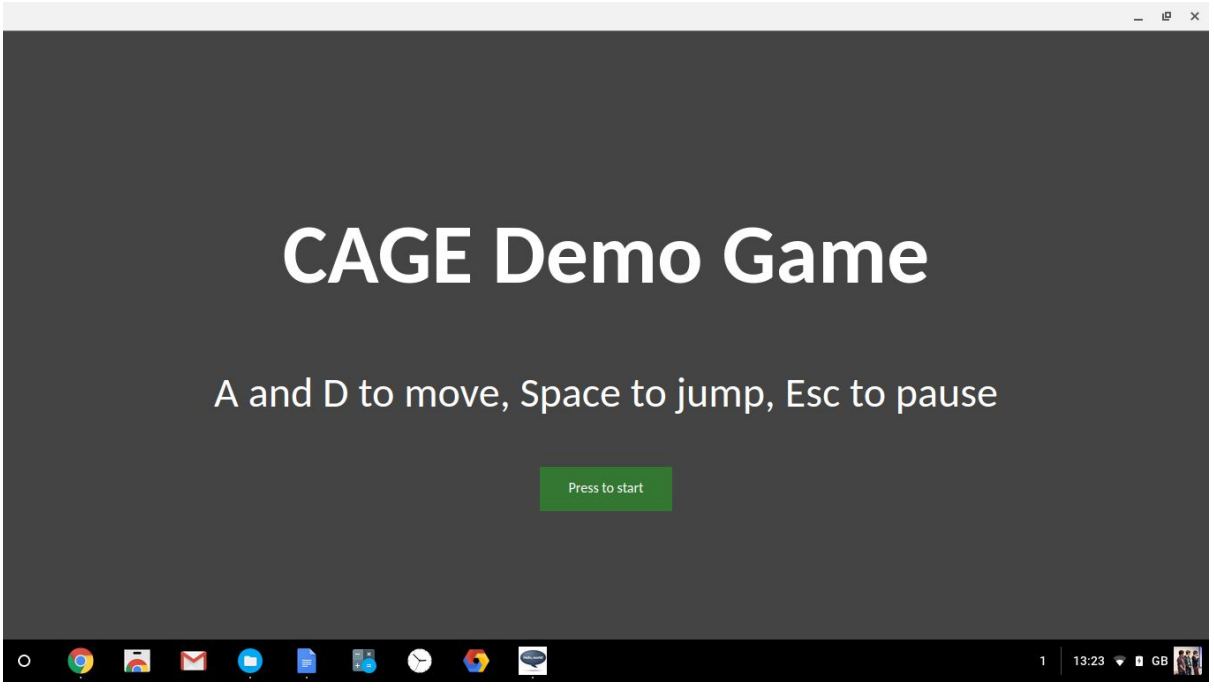
I have tested my programs in 3 key ways:

- I played through the game and tested each of the game elements to ensure they worked correctly.
- I had my initial users playtest my game to look for bugs as well as to comment on its quality.
- I had my initial users build a simple game using a tutorial I wrote in order to receive feedback on the game engine itself.

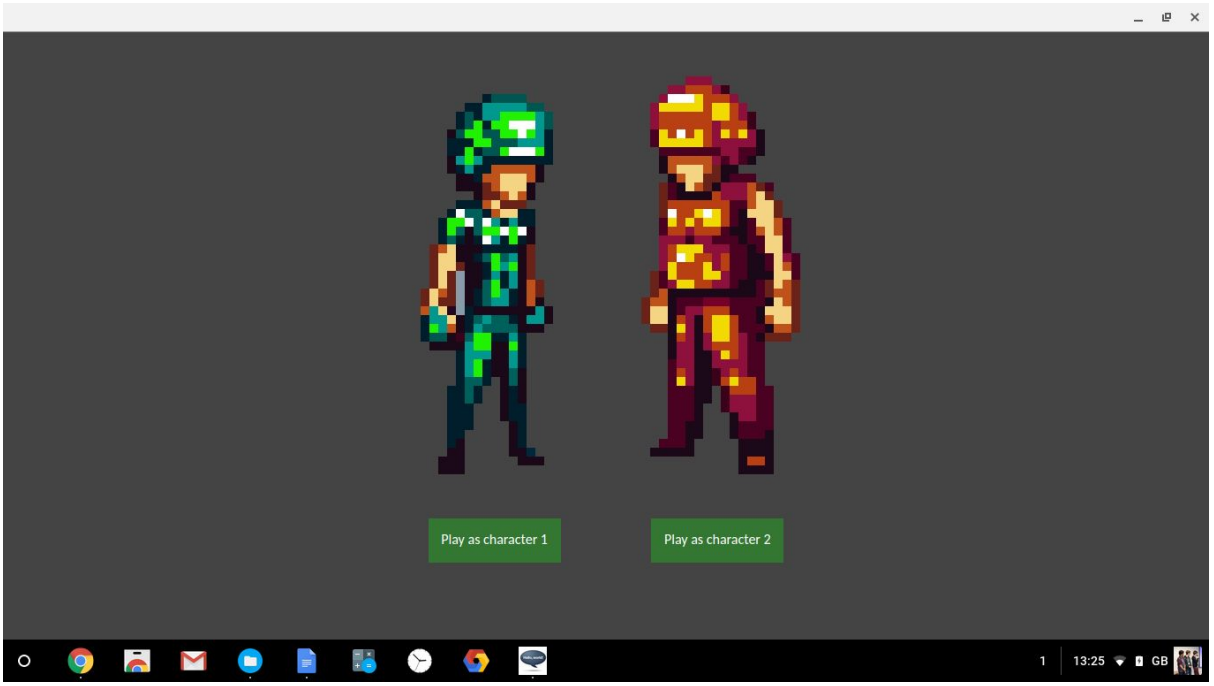
Video Testing:

The video I recorded of me testing the game is available as an unlisted YouTube video:

Screenshots:

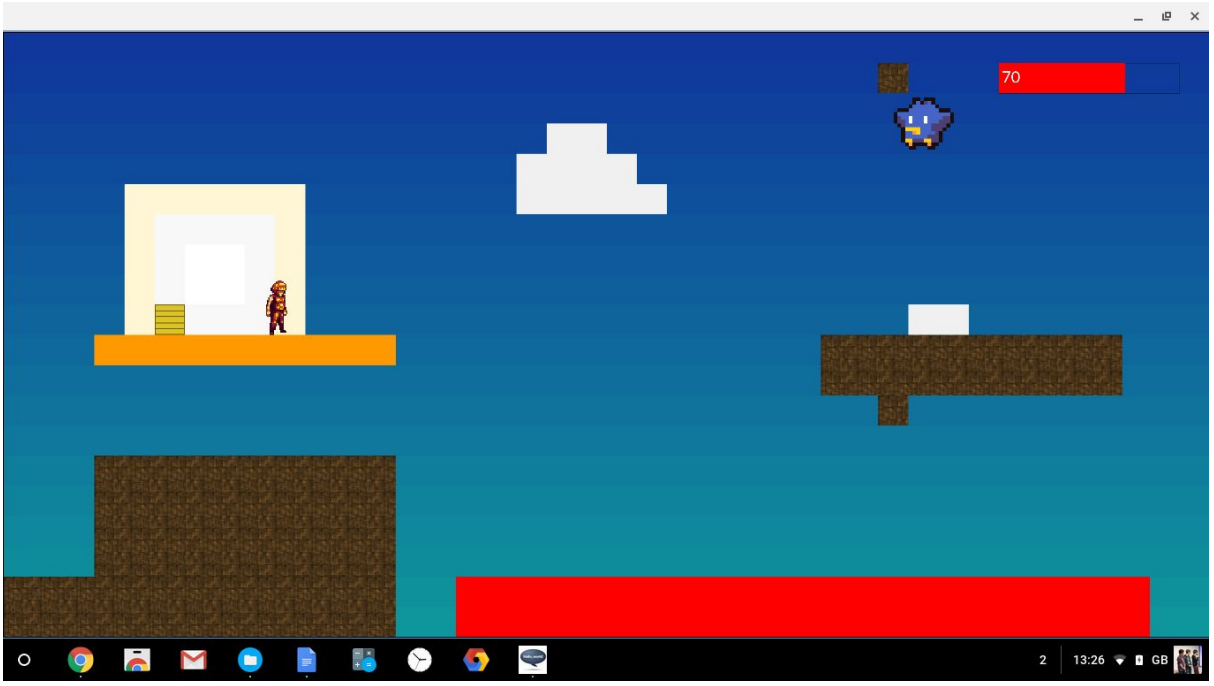


The initial menu

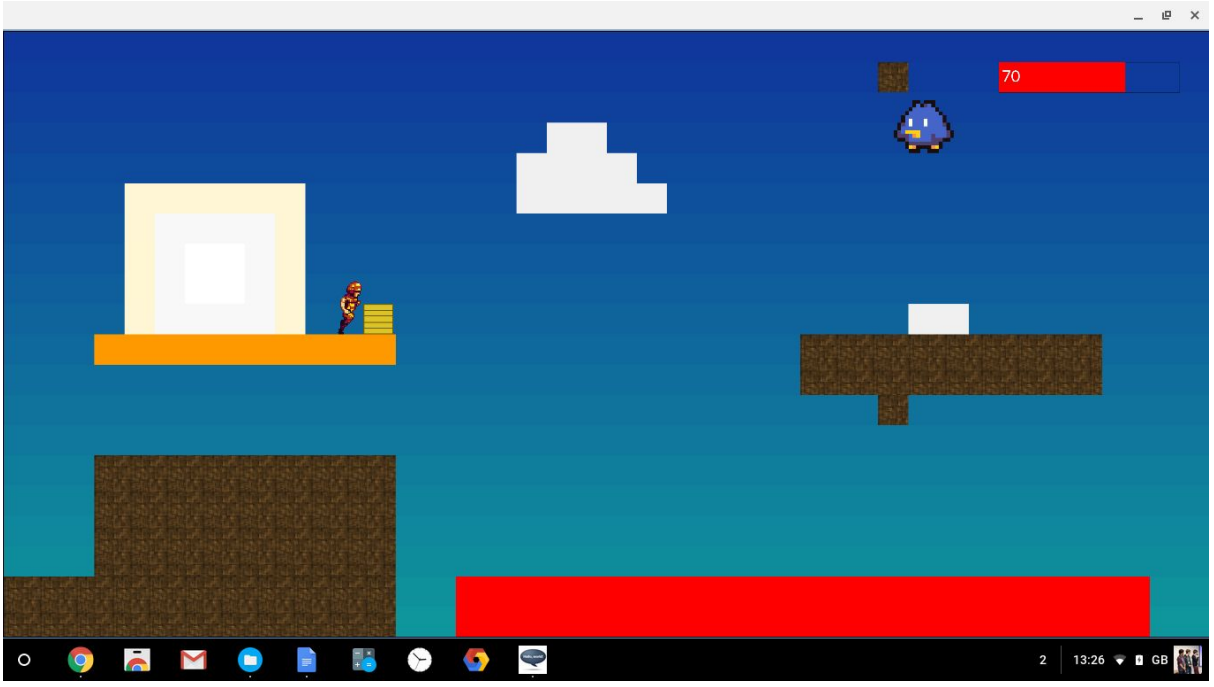


Character Selection Menu

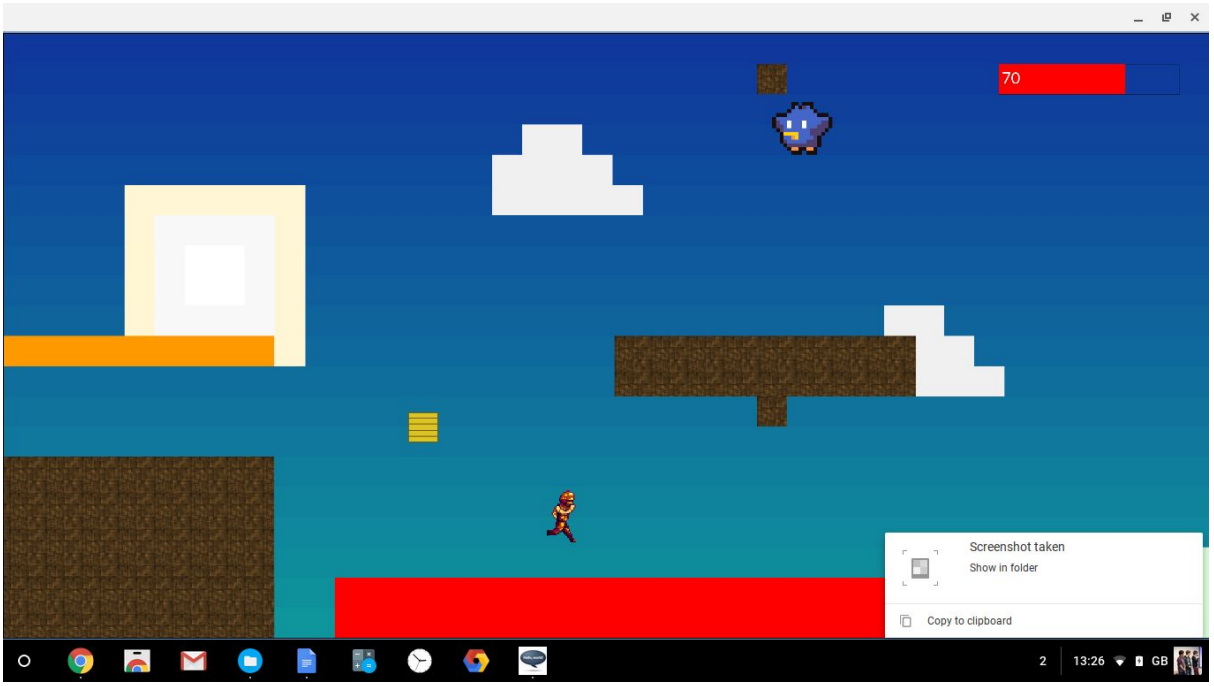
Initial Playground Level with character 1



Initial Playground Level with Character 2



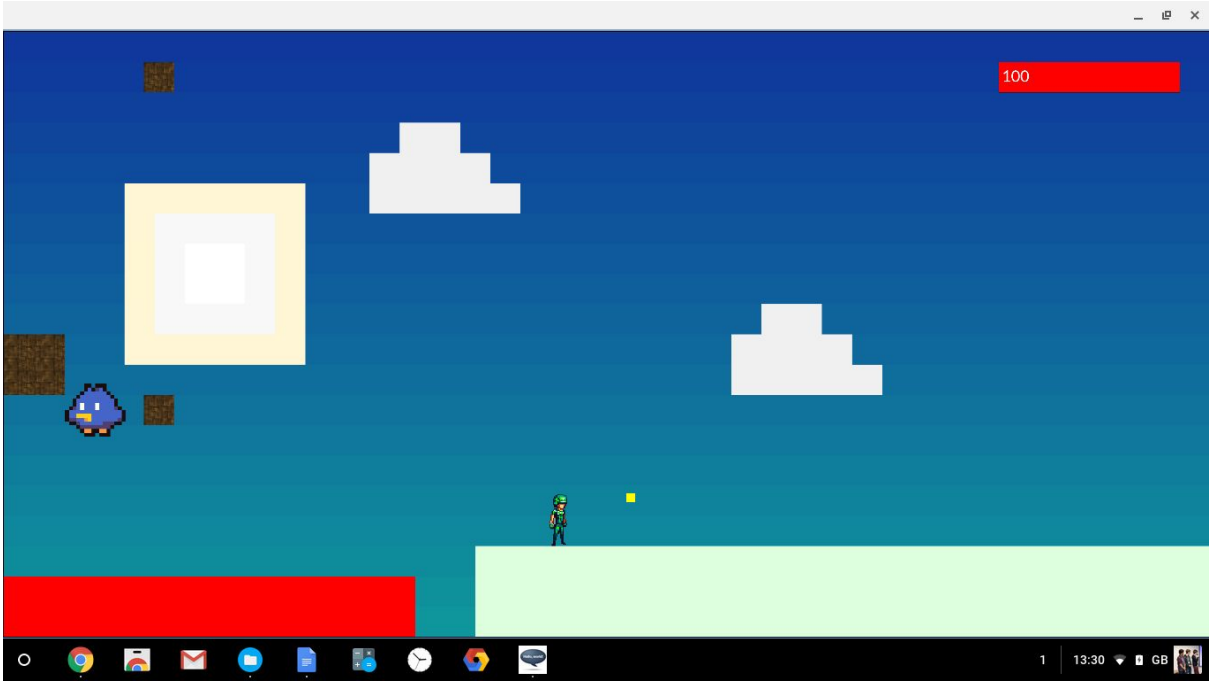
Character 2 pushing a crate



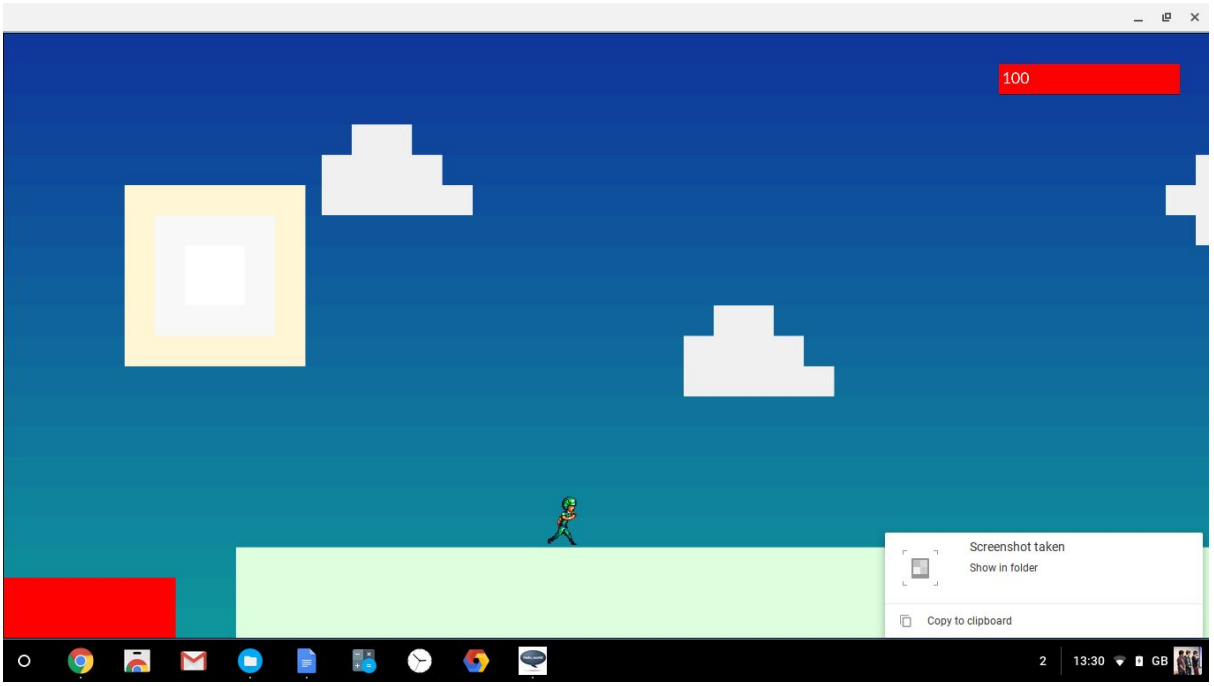
Character 2 and a crate bouncing off a red bouncy platform.

Character 2 and a crate on a moving platform.

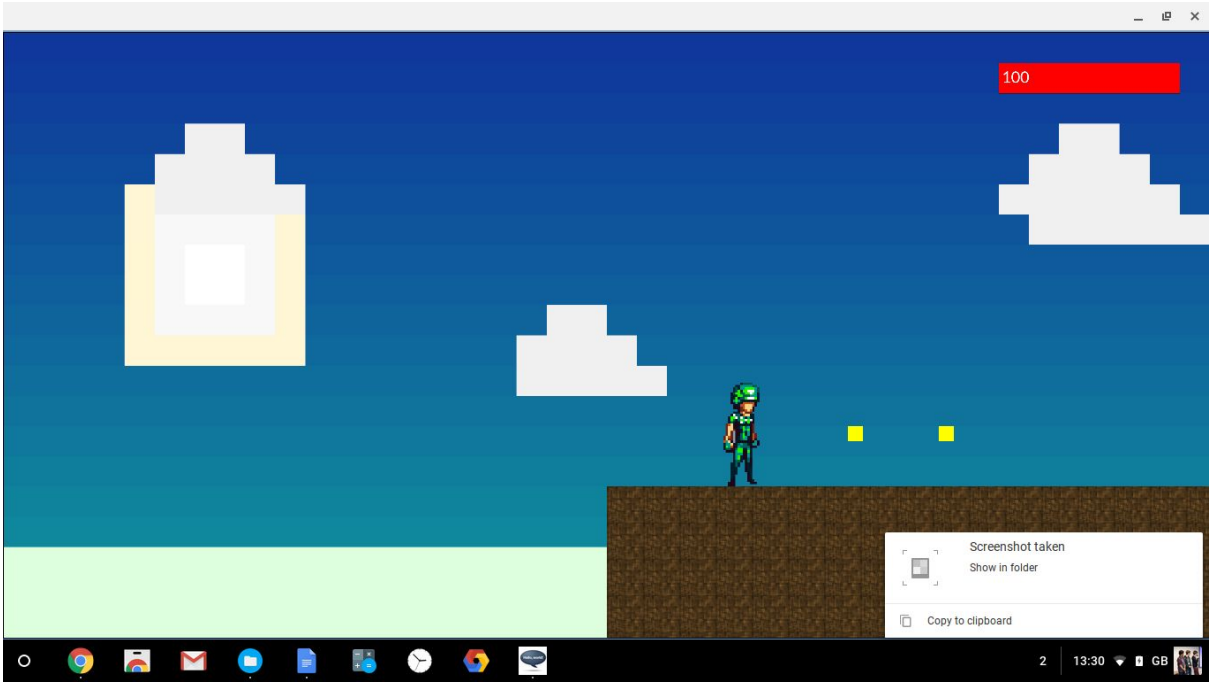
A vertical flying enemy is caught and moved by the moving platform.



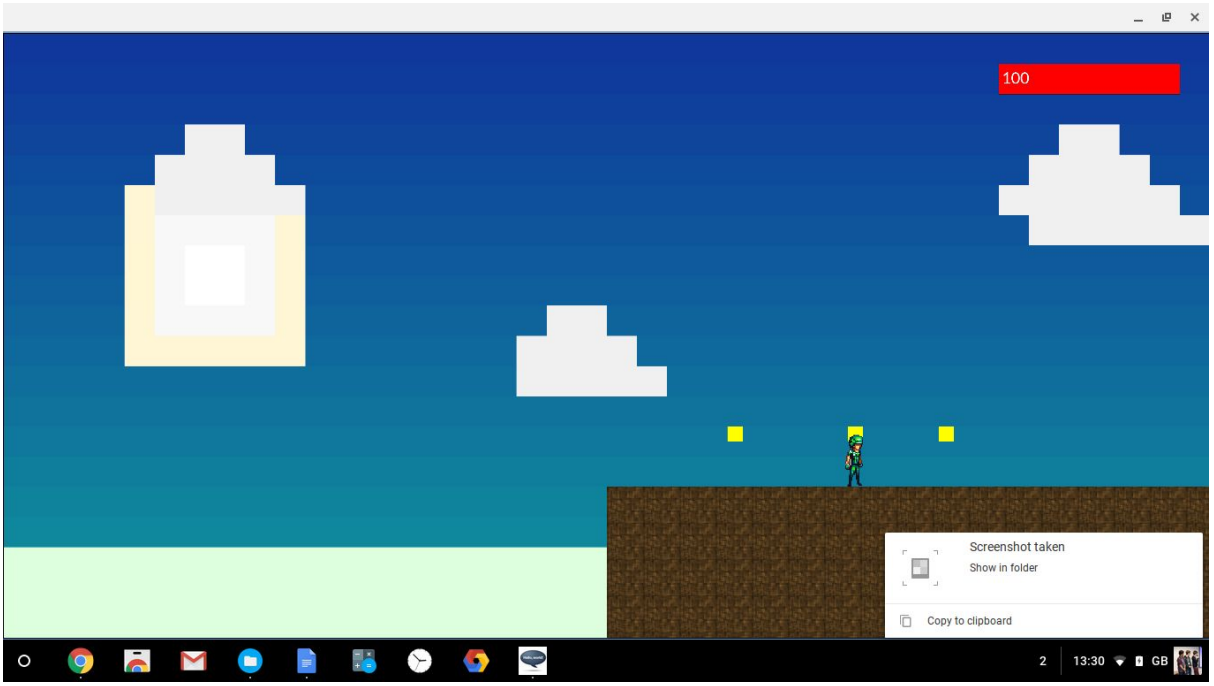
Character 1 on Ice, next to a coin.



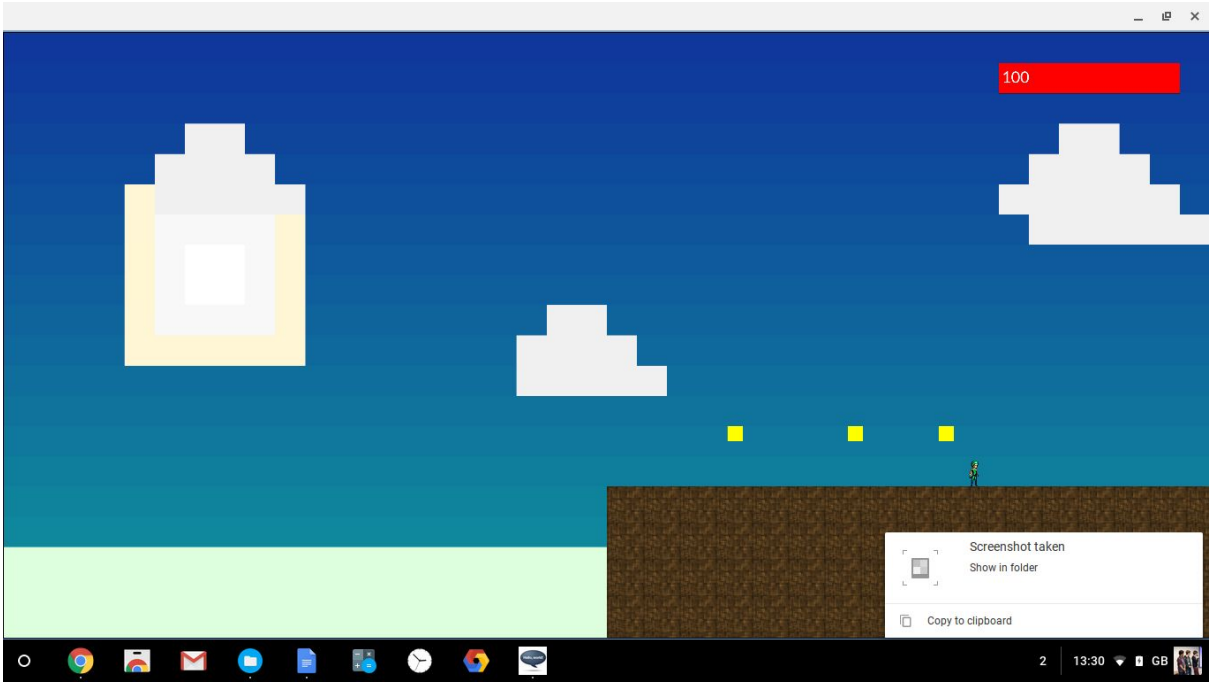
Character 1 running on ice, having collected the coin.



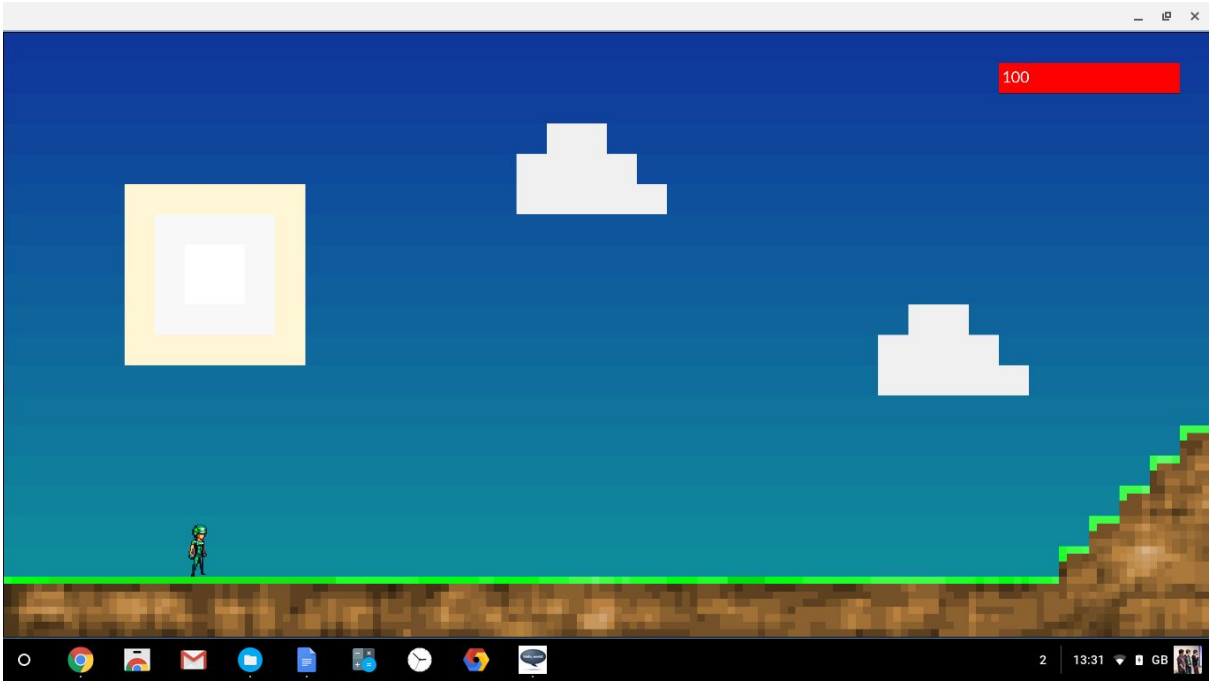
Character one, having touched the 2x size powerup.



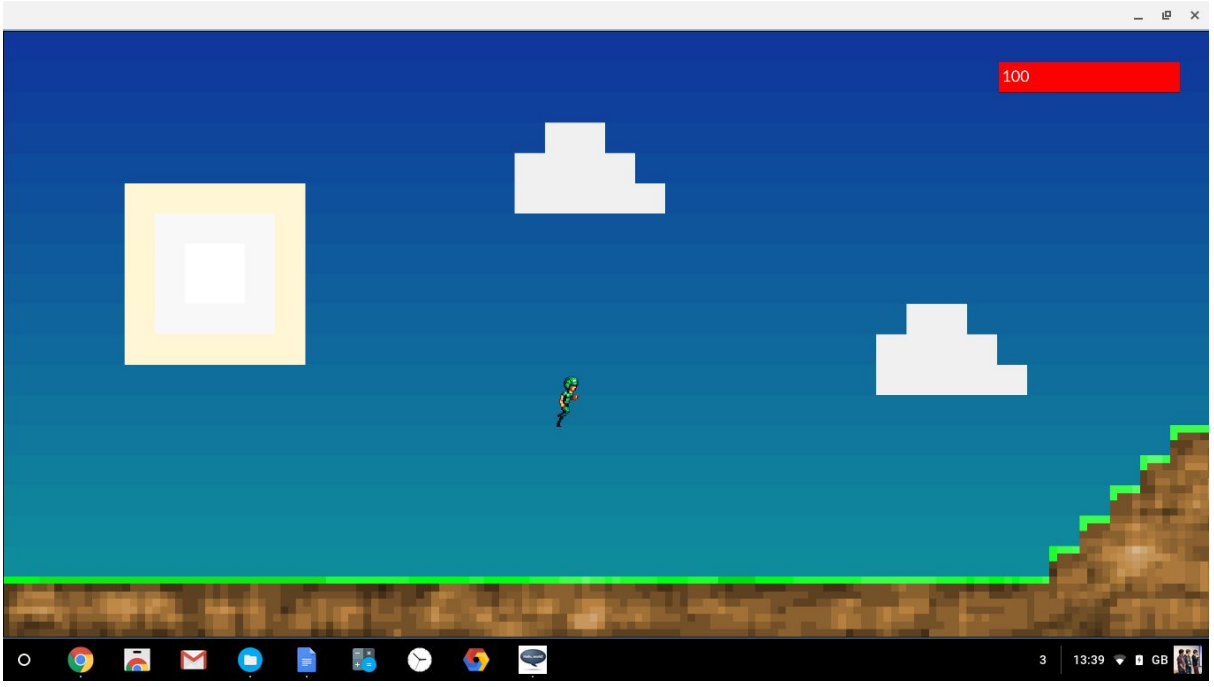
Character one, having touched the 1x size powerup.



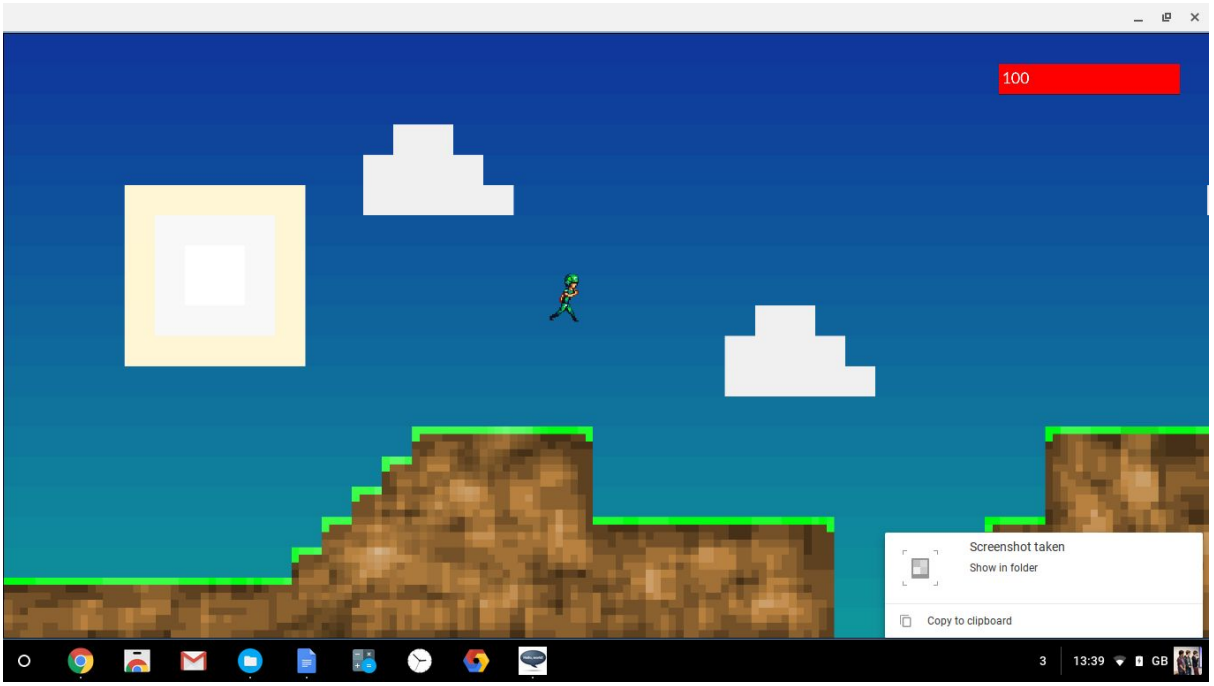
Character one, having touched the 0.5x size powerup.



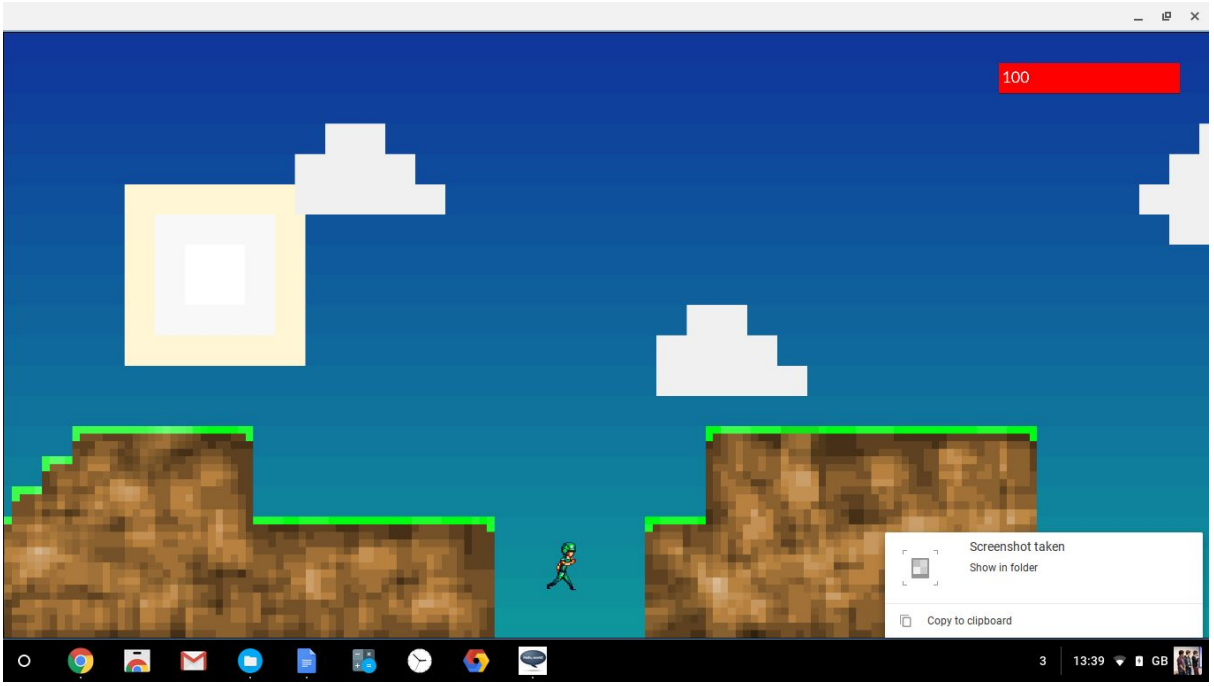
Character 1 at the start of level 1.



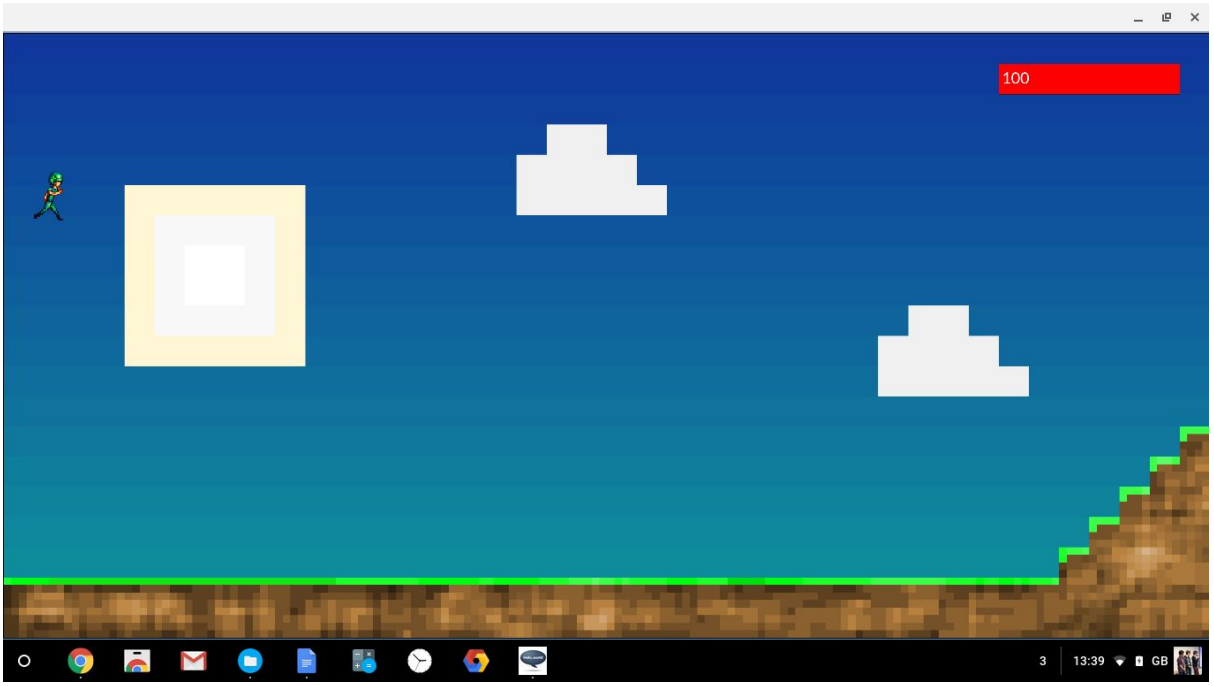
Character 1 jumping across the level, showing how the level does not scroll unless the player is within the active zone.



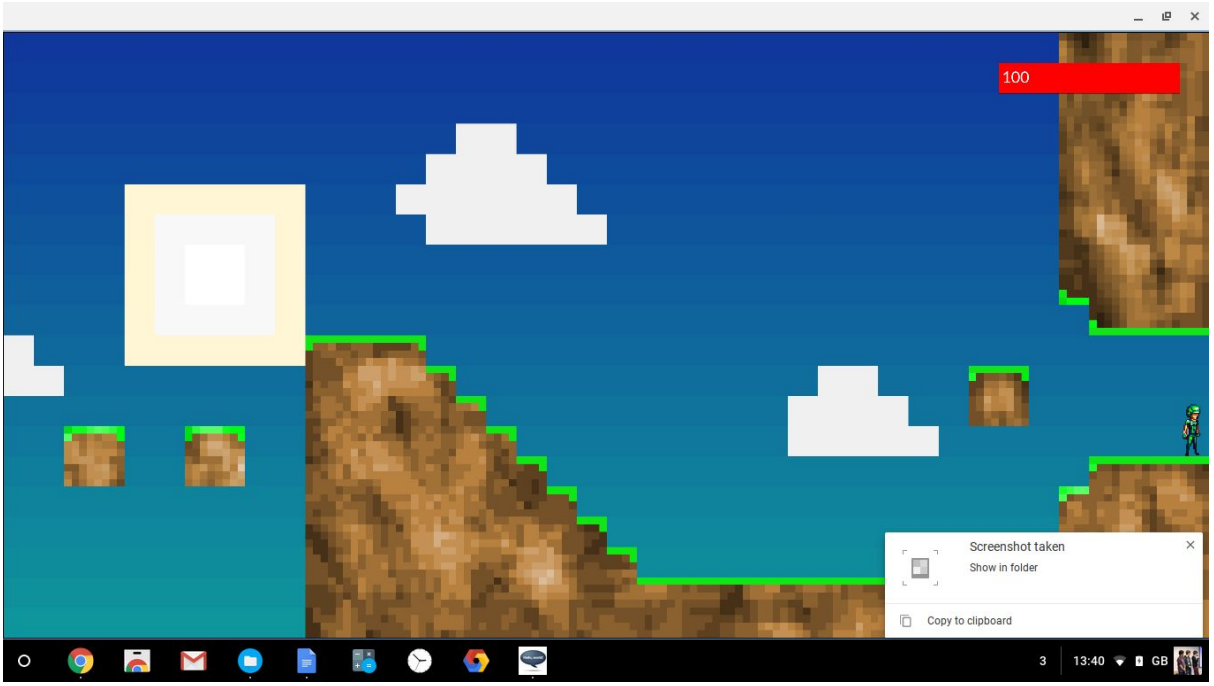
The player jumping up a set of stairs, scrolling the level for the first time.



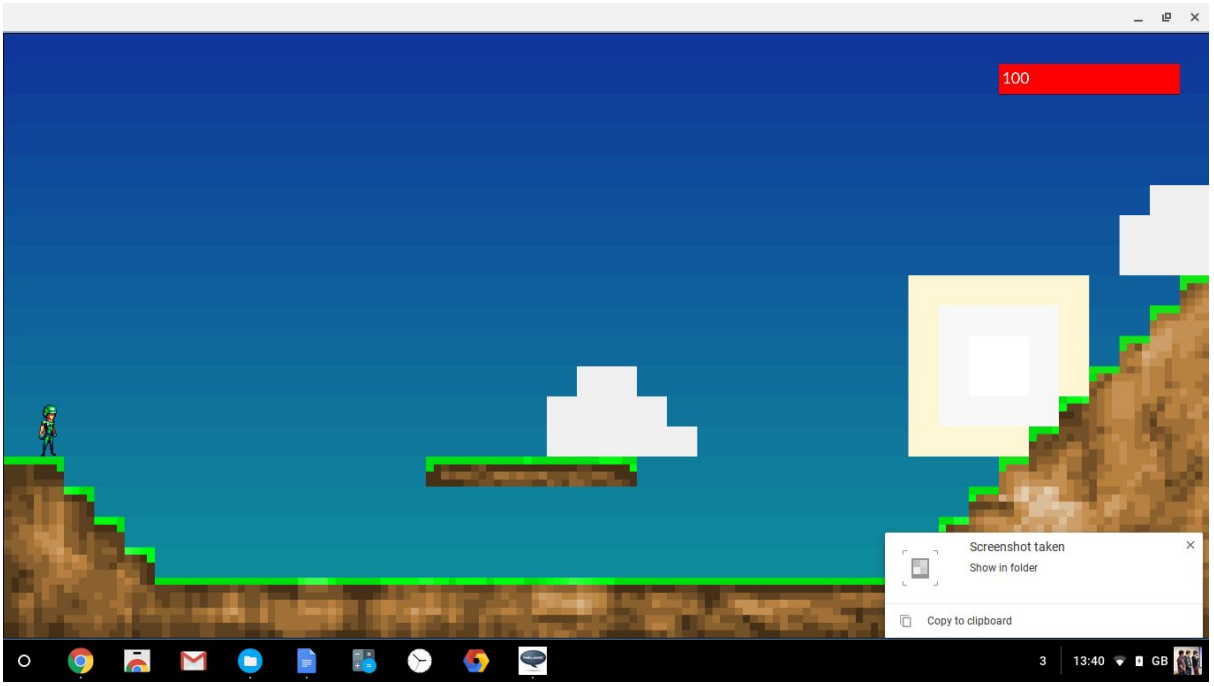
The player falling into a pit



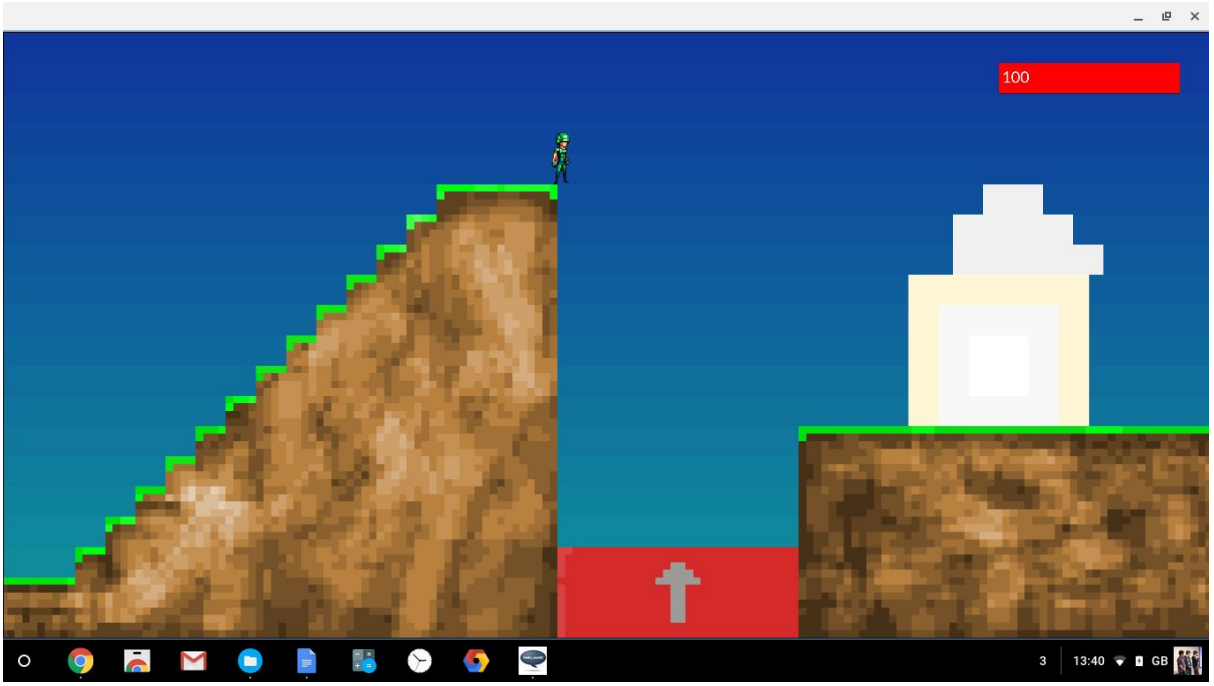
The player being reset after dying.



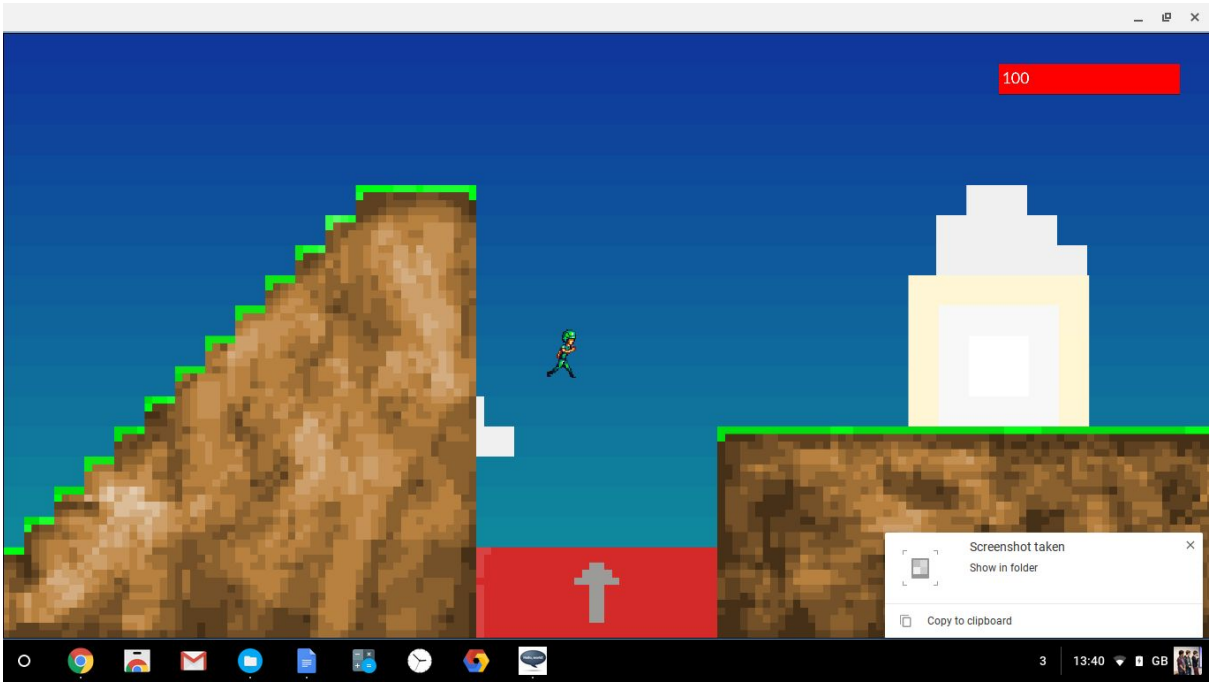
The player reaching the end of the first level



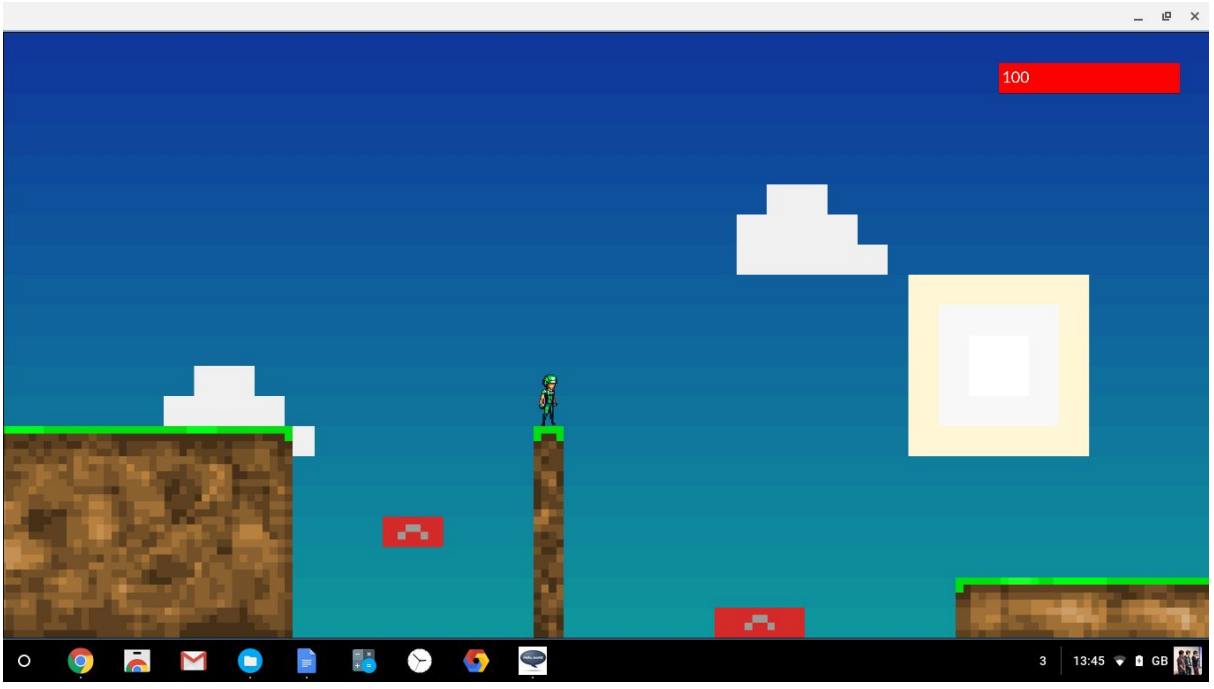
The player, having moved to the right and offscreen, is transported to the second level.



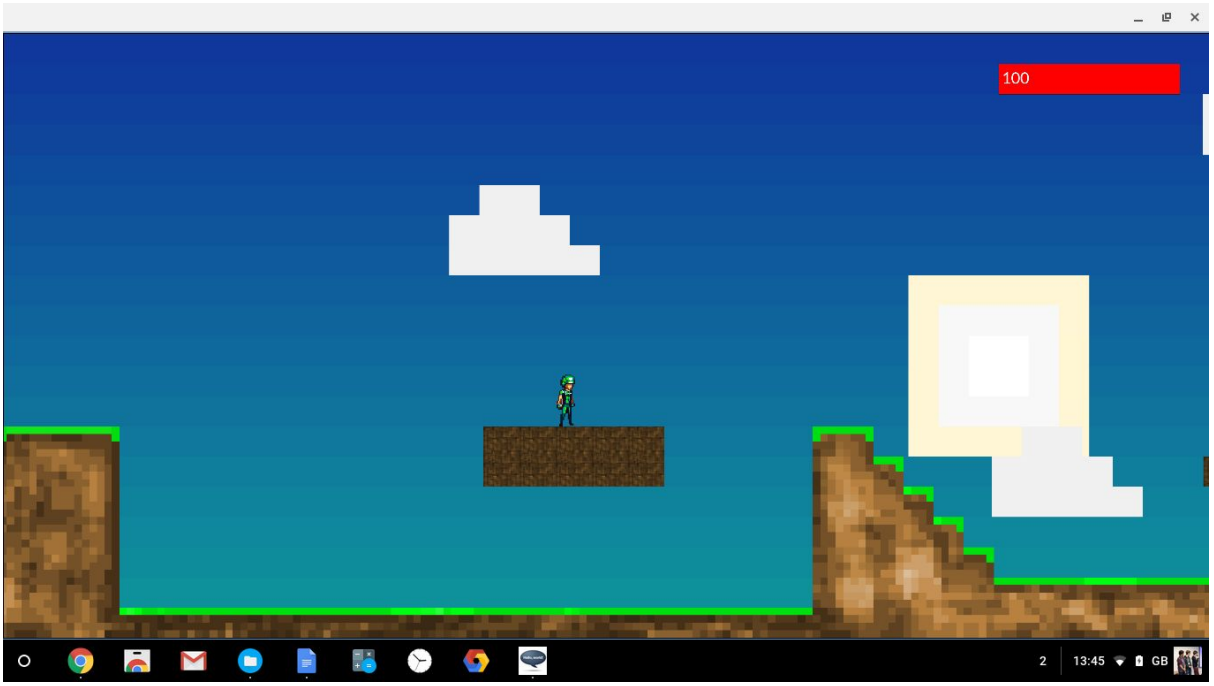
The player standing above a textured bouncy platform.



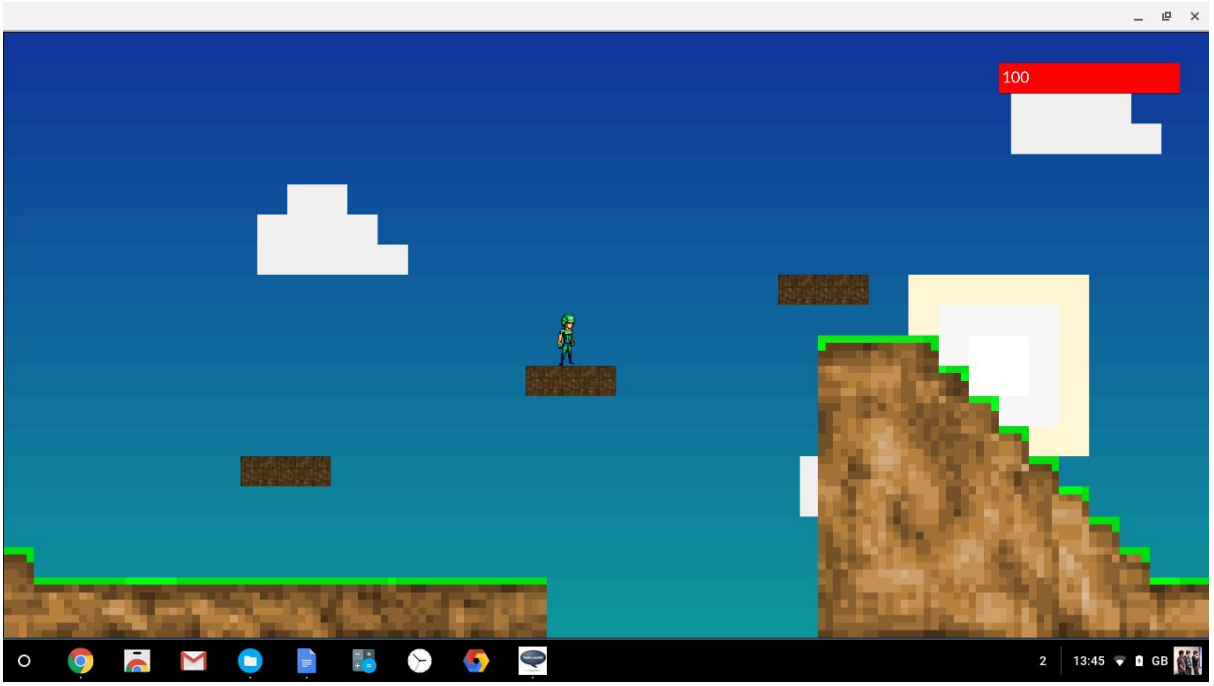
The player, having bounced off the platform does not need to jump to reach the next platform.



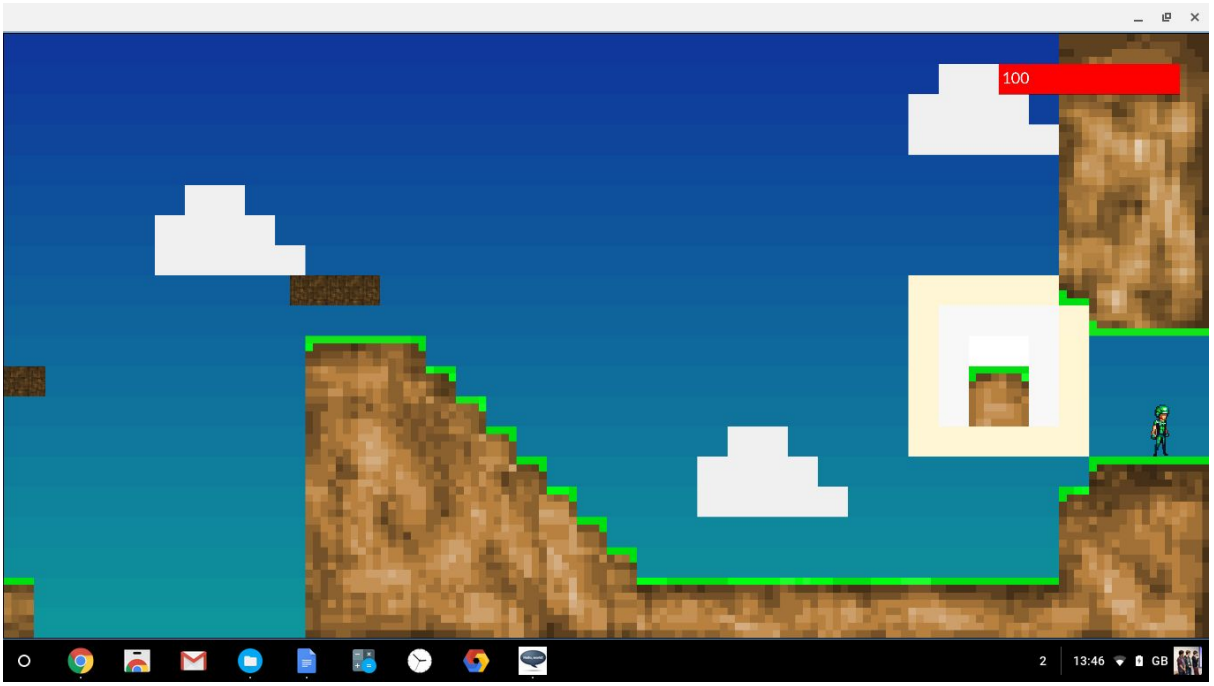
An example of how bouncy platforms can be used to create interesting game elements. The player can run from the left to the right without needing to jump at all.



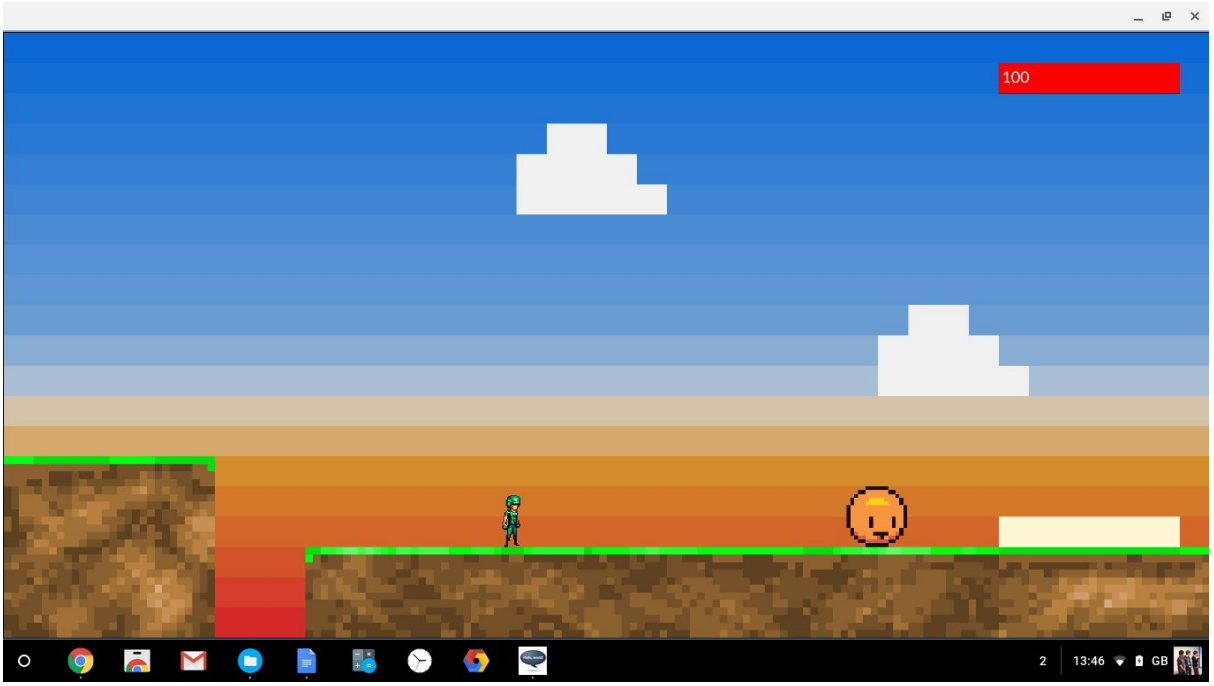
The player on a moving platform.



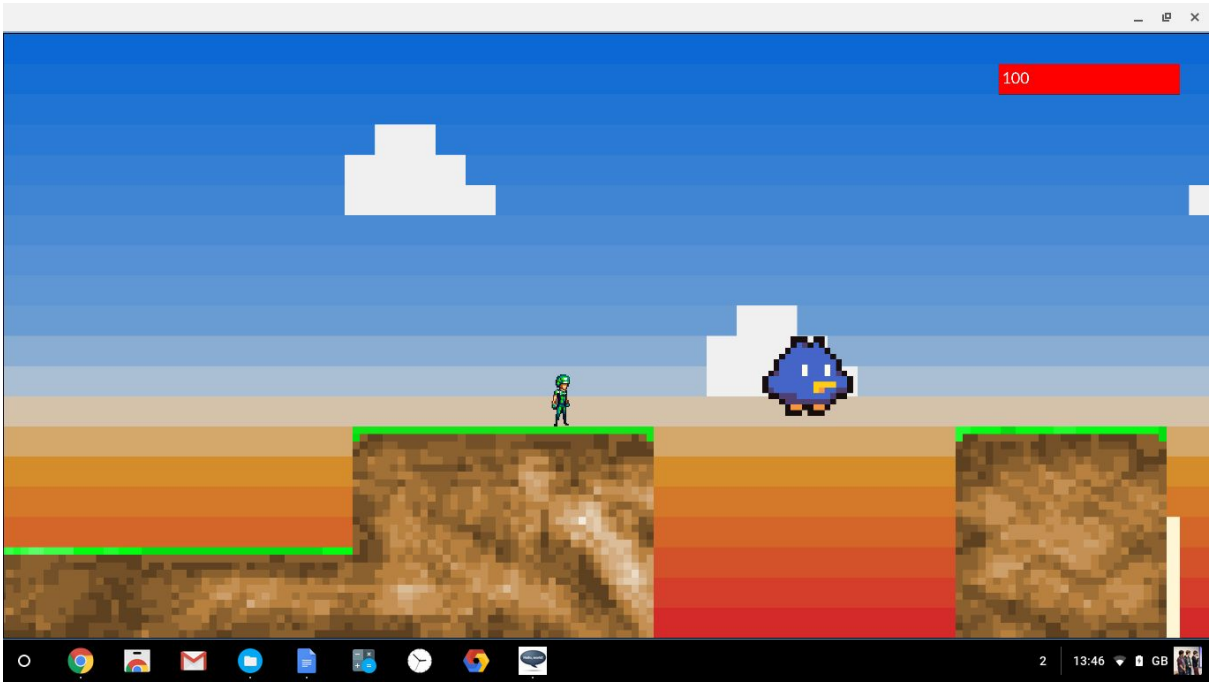
More moving platforms near the end of the second level. The way these move relative to each other add a level of difficulty



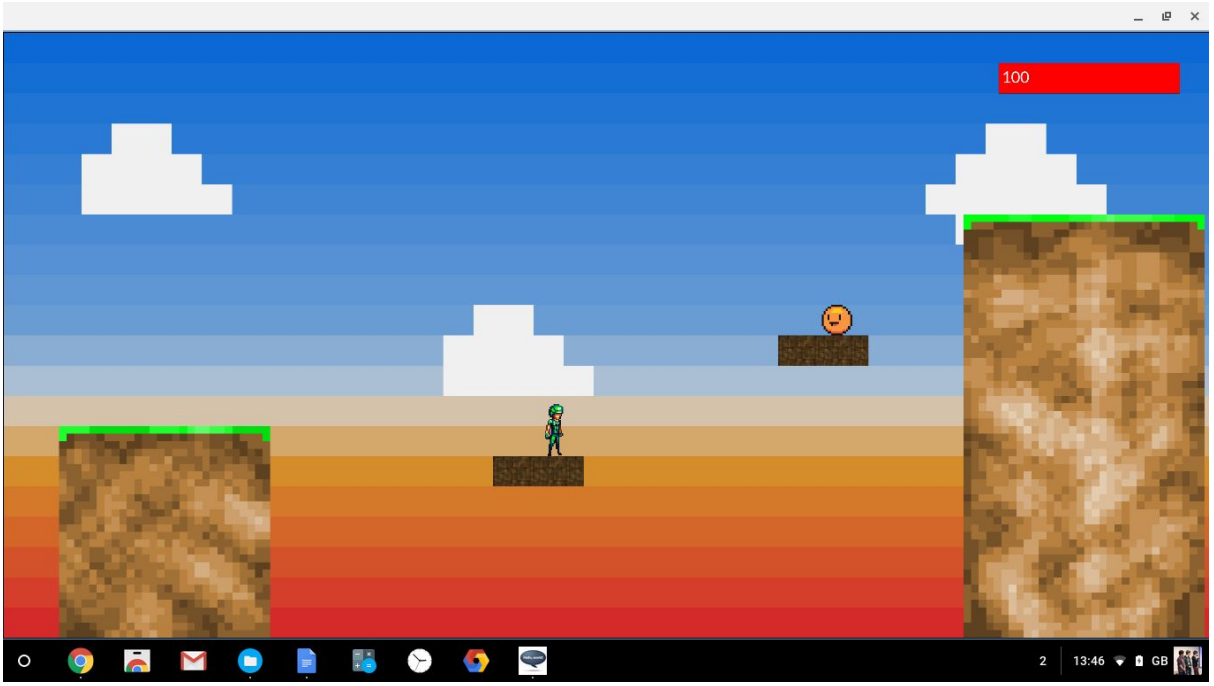
The end of the second (and third) levels are the same to show the end more easily



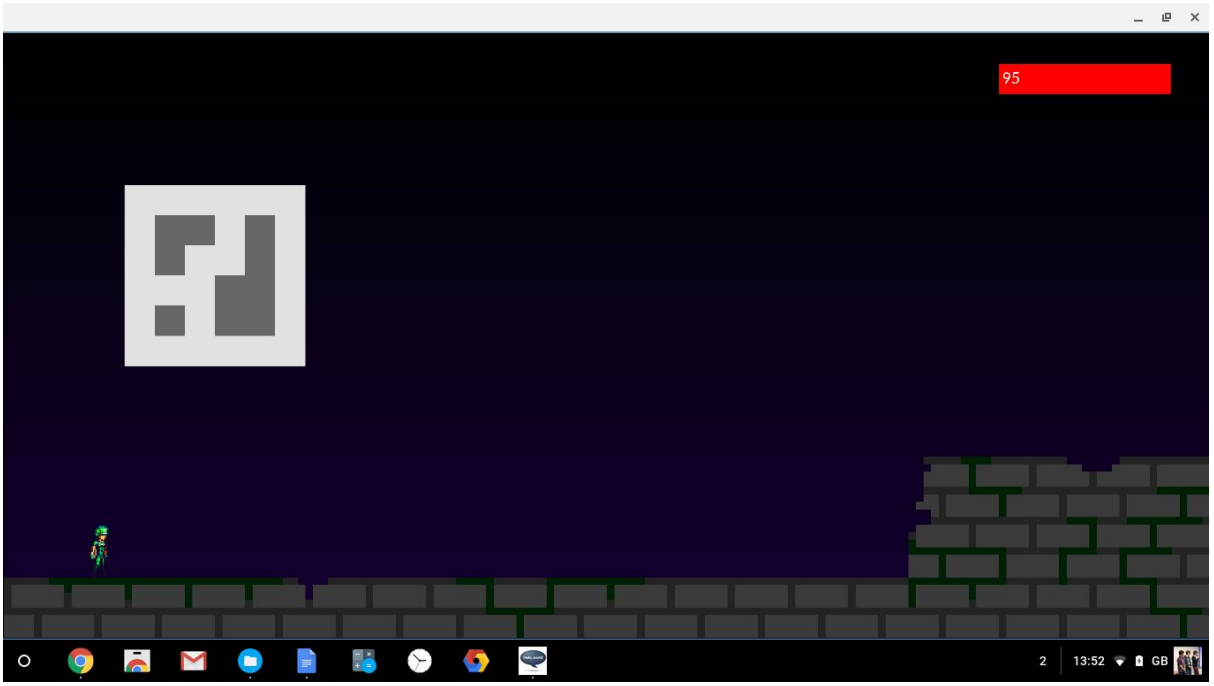
The start of the third level, with a simple horizontal patrol enemy



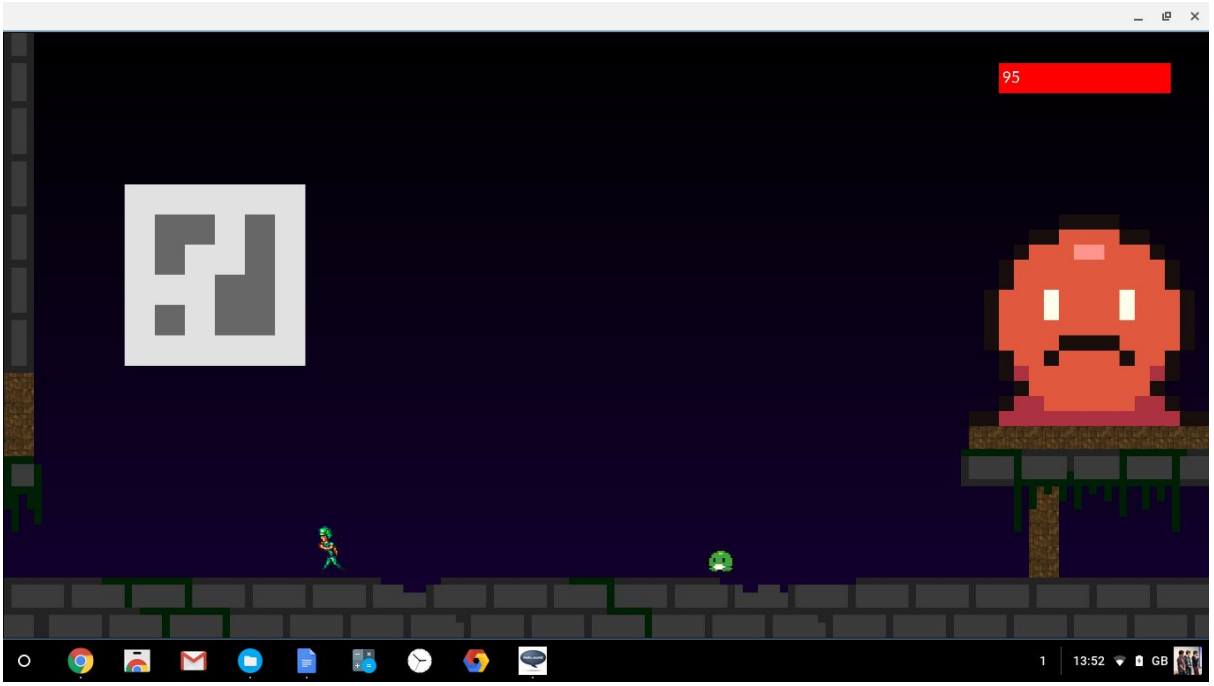
A flying patrol enemy.



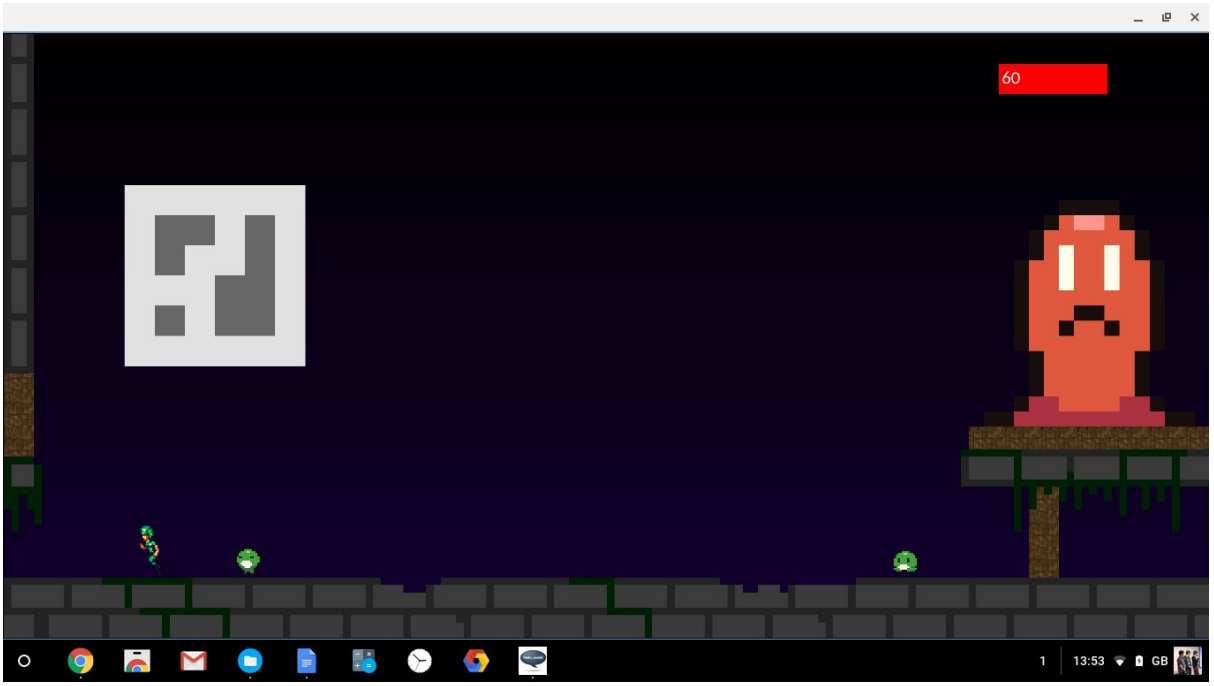
The player and an enemy on moving platforms.



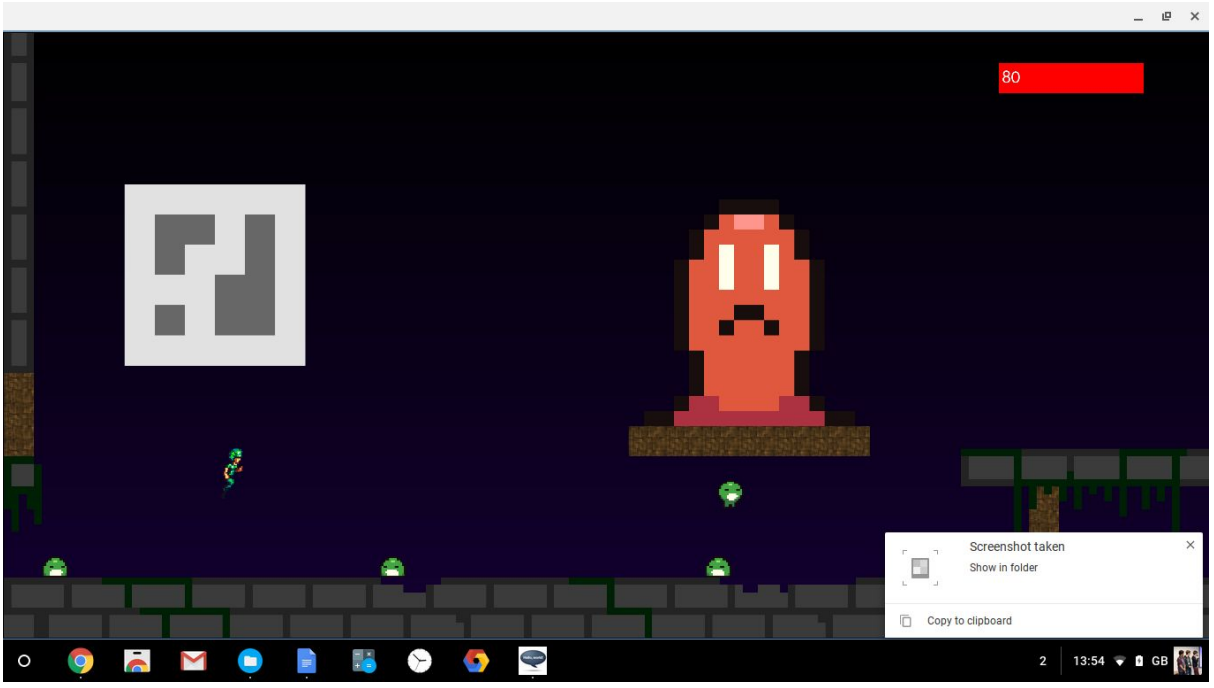
The start of the final level



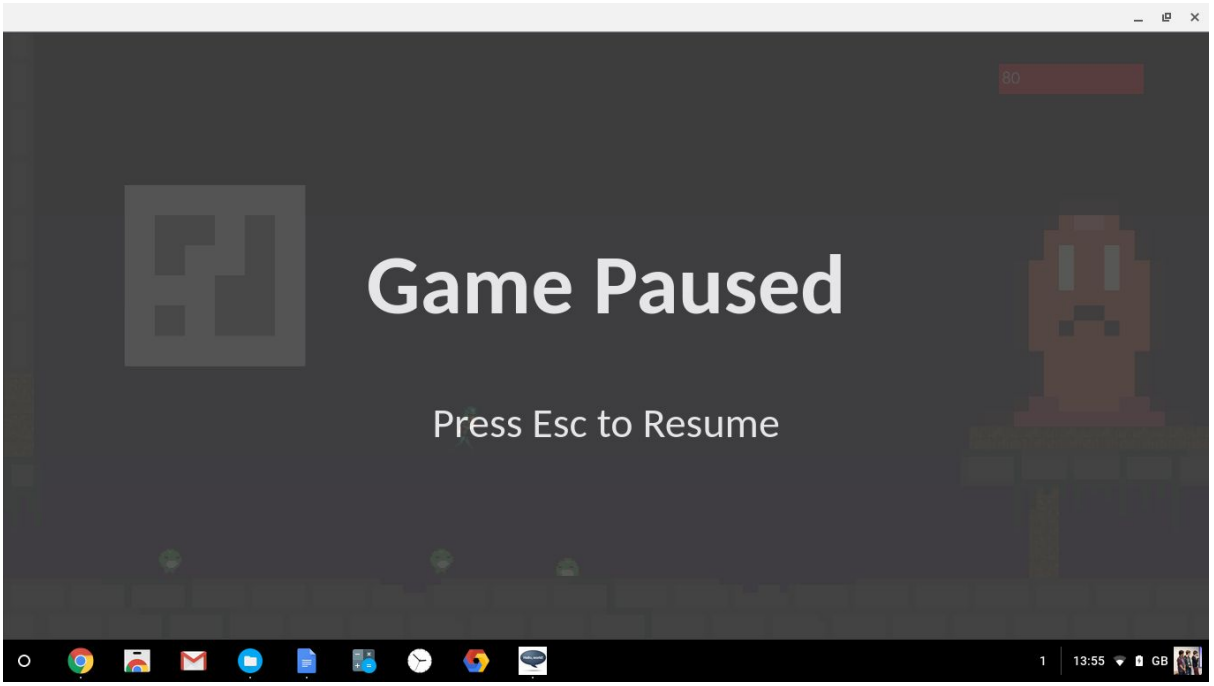
The boss, which has released a small patrolling enemy



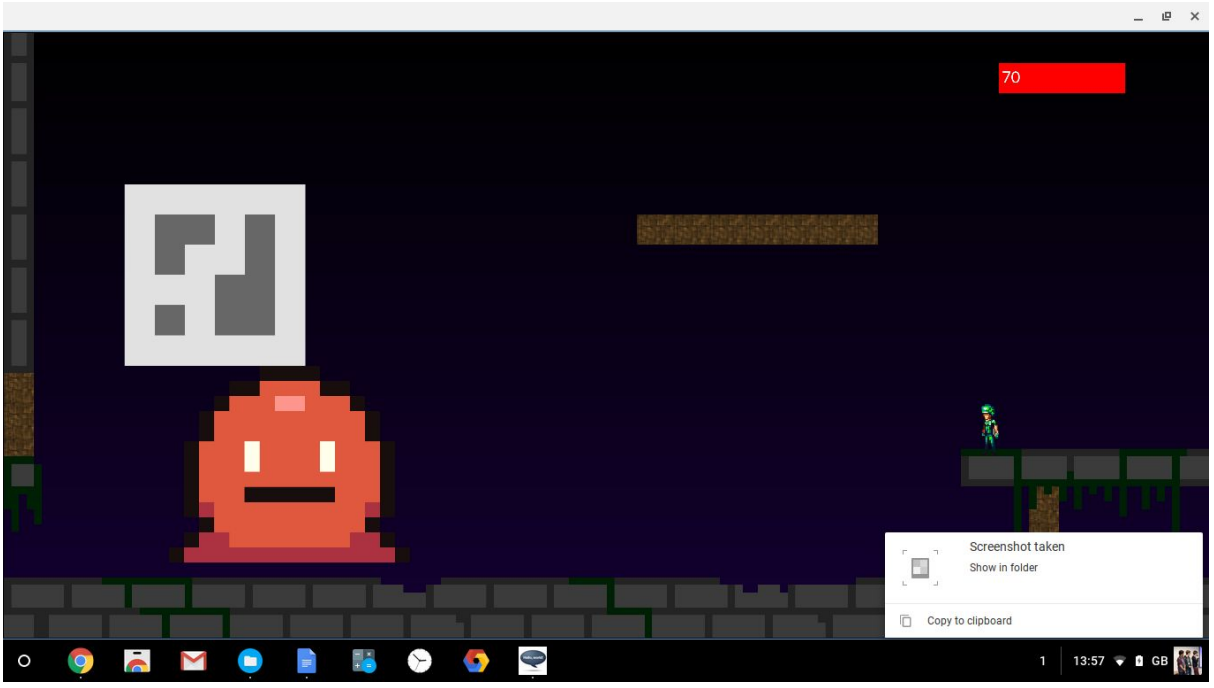
The boss releases more enemies to test the player, each slightly faster than the last.



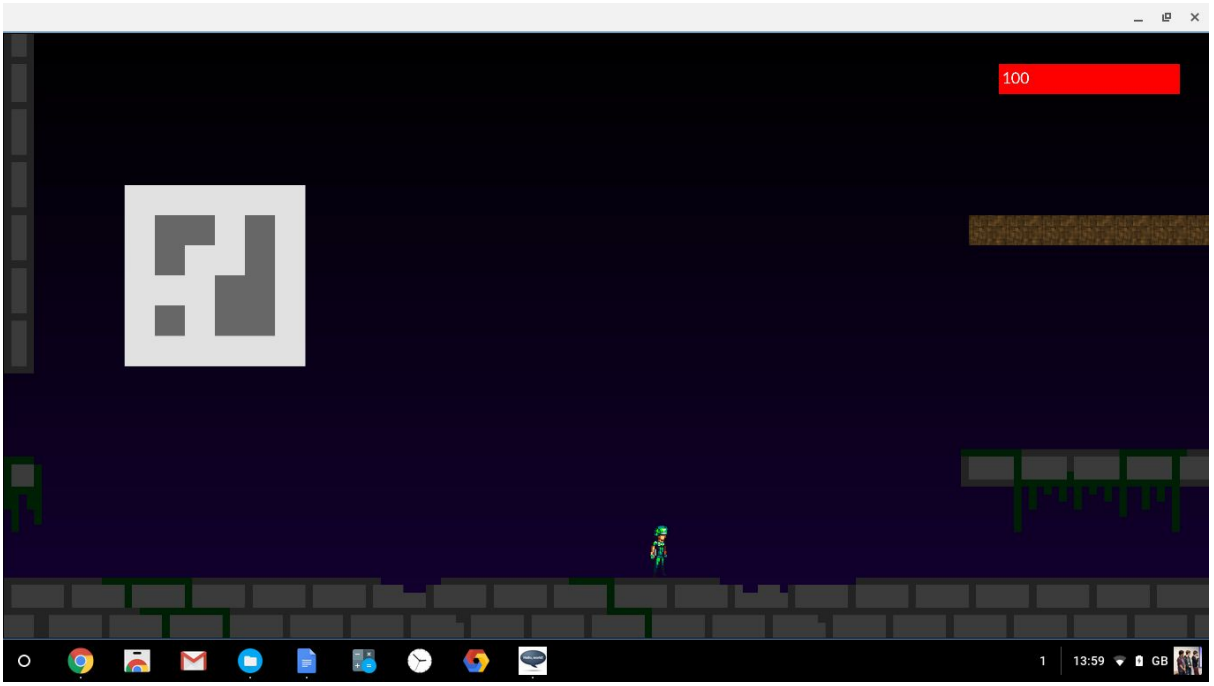
The boss flies out on the moving platform once the player destroys all the mobs it throws at them. The flying boss releases 8 more mobs in an attempt to kill the player.



The pause screen, accessed by the Esc key



Once the player defeats all mobs, the platform returns to its original position, rises up and moves back and forth at a higher level. The boss then jumps off and moves back and forth below, jumping randomly, trying to crush the player.



The doors to the left and right open when the boss is killed, and the platform stops moving.

User Feedback:

Game

Q) What are your first impressions of the game?

Hugh:
The graphics aren't the best I've ever seen, if I'm honest, but the game is smooth, and there aren't any bugs, which is good. I really like how native the app looks. Is the [player] animation changeable? I think it needs more frames.
You would need to modify the PlayerSprite function, but yes.

Matt:
You're not a Graphics Artist are you? These are poor.
No, I've never really drawn anything before.
I mean as long as users are better at making graphics than you are, I don't see any problems with the game. The player movement is crisp, the level scrolls well, the boss fight is challenging, I just wish it was longer.
I'm considering actually making a game using this engine, instead of simply leaving it online.
I think if you actually put some time into level design and character design, it would be very good.

Sam:
I enjoyed playing with the features in the playground level, I wish the actual game used more of the features. Some of the timings of the platforms and mobs could do with some adjustment, I've been waiting for them to come within range quite a lot, it's annoying,
Aside from my poor game design, does the game run well?
Oh, yes, the game runs very well, there aren't any frame drops, all the animations are smooth, the player moves as i expect it to, its pleasing to play, if a little short.

Q) Is there anything you particularly like about the game?

Hugh
I really like the parallax background, even with your graphics it looks very cool.

Matt
The way the level scrolls when you're on a moving platform is quite nice, I like the way things scroll relative to you.

Sam
The little playground level at the start is handy, I like the way it sort of throws all of the features at you so you can experiment with the features of the game before you get into it.

Q) Is there anything you particularly dislike about the game?

Hugh

Besides your graphics?
Preferably
I think the animations could be improved, allow game elements have animations, and make the number of frames and the speed variable.

Matt
The levels are quite short, I'm getting through them in under a minute. I think the game needs to be longer, there's not enough content here to fully show off your game.

Sam
The graphics need a lot of work, the player and mobs need more animation frames. The player moves smoothly, but It accelerates too fast I think.

Q) Are there any features or improvements you would like to see added?

Hugh
It would be good to see more of the modules used in the game, with more levels, better graphics and longer levels.

Matt
Higher resolution graphics, more levels, and the use of more of the modules you have written!

Sam
I think you need to write more modules, you've made enough to show off your game engine, but more modules would be very cool, perhaps a powerup that modifies gravity, or cannons, or enemies that move in more complex ways?

Engine

Q) Can you try following the tutorial. How do you find following it? Is the Engine Logical and simple to use?

Hugh
Your tutorial is simple enough. Downloading all of the files was annoying but I assume once the user knows their way around the system they can just copy and modify the DemoApp?
Yes, that was what I intended.
The engine is quite simple to use, even though it does assume basic programming knowledge. All of the constructors are logically defined, and the system does fit together well, even if the dependency system isn't the easiest to grasp.

Matt
If I didn't read into how your game engine works, the downloading files stage of the tutorial seems kind of meaningless

Sam

The graphics need a lot of work, the player and mobs need more animation frames. The player moves smoothly, but It accelerates too fast I think.

Q) Can you try adding the ice module in, and using it in your level? How was the process?

Hugh
[Needed no help adding or the module] I think that was a simple enough process, though perhaps add a little readme.md file onto github with the constructor and its arguments in.

Matt
[Needed help navigating the repository but added module in correctly once located] That process was simple enough, though perhaps you need to document it in the modules folder in case someone forgets how to include a script file

Sam
[Needed help including the script file within the index file, but could do all other steps] I think you need a more in depth tutorial on that process, I have no idea what adding that line to index.html did. I did find using the module once i added it easy enough.

Q) Do you find the interface easy to use? Are there any improvements I could make to the system?
Hugh
Yes, it's easy enough to use, though I shouldn't really need to check the source to learn what the constructor takes as an argument. You need to add those readme files I mentioned

Matt
Your repo could do with a little explanation of where everything goes together with the tutorials and the modules, I had difficulty finding where the Ice module was. Aside from that, i do like the way the constructors work, it makes the level files surprisingly small.

Sam
I think the system is kind of hard to grasp, though I know nothing about web technologies so perhaps that's why I struggled. Your documentation is good though perhaps you need a page with a few links to websites to catch up to the baseline skill level you need to use the engine

Evaluation

The aims I set out to complete were:

- Create an ES6 based modular 2D game engine
- Create a set of demo modules to work with the engine
- Create a basic game as a proof of concept of the system working
- Create a centralised resource for the game engine, containing tutorials and documentation on each of the modules I write.
- Create a ES6 inheritance based module structure, utilising polymorphism and aggregation to create small, lightweight modules that can be used to add functionality to games.
- Create a sample, single level project aside from the basic game as a project to demonstrate how to create and edit levels
- I will fully document every method and property of every core module in order to make the system user friendly and accessible.
- I will add a method of detecting when certain events have occurred in order to trigger certain user-defined behaviours.
- Write a set of logical constructors for the users to use when creating levels.

And I believe I have met most if not all of these targets. My Engine is 2D, Modular, and uses ES6. I wrote a set of Object-Orientated modules, which I used to create levels for my sample game, as well as the demo game level. Both of these are uploaded to a Github Repository along with all of the modules, and all of the engine code. The repository also contains documentation for all of the core modules, as well as a brief explanation of the engine, and a tutorial.

If I were to continue development of this engine, I would likely add many more modules, write a level editor, and continue expanding my game or even write multiple to show off the features of the engine.

I would also write more documentation on all of the code, explaining it in more detail. Perhaps by using a more in-depth documentation system.

Appendix

CAGE Files:

background.js

```
/**
 * Listens for the app launching, then creates the window.
 *
 * @see http://developer.chrome.com/apps/app.runtime.html
 * @see http://developer.chrome.com/apps/app.window.html
 */

chrome.app.runtime.onLaunched.addListener(function() {
  var windowWidth = screen.availWidth;
  var windowHeight = screen.availHeight;
  chrome.app.window.create('index.html', {
    id: "mainWindow",
    outerBounds: {
      width: windowWidth,
      height: windowHeight,
      left: 0,
      top: 0,
    },
    resizable: true,
    alwaysOnTop: false,
    frame:"chrome"
  });
```

debug.js

```
//Get the HTML elements to output debug data to
var xSpan = document.getElementById("x"),
    ySpan = document.getElementById("y"),
    vxSpan = document.getElementById("vx"),
    vySpan = document.getElementById("vy"),
    ijSpan = document.getElementById("ij"),
    loSpan = document.getElementById("lo"),
    lxSpan = document.getElementById("lx"),
    fpSpan = document.getElementById("fp");

//Output debug data every frame
function debug() {
  xSpan.innerText = game.players[0].x;
  ySpan.innerText = game.players[0].y;
  vxSpan.innerText = game.players[0].vX;
  vySpan.innerText = game.players[0].vY;
  ijSpan.innerText = game.players[0].isJumping;
  loSpan.innerText = game.levels[currentLevel].offset;
  lxSpan.innerText = game.levels[currentLevel].offset + players[0].x;
  fpSpan.innerText = Math.round(1000/delta);
}
```

game.js

```
//Request Animation Frame Shim/Polyfill
(function() {
    var requestAnimationFrame = window.requestAnimationFrame ||
window.mozRequestAnimationFrame || window.webkitRequestAnimationFrame ||
window.msRequestAnimationFrame;
    window.requestAnimationFrame = requestAnimationFrame;
})();

game = {
  debug: false,
  paused: true,
  levels:[],
  players:[],
  hud:[],
  pause: function() {
    game.paused = true;
  },
  play: function() {
    game.paused = false;
  },
```

```
  kill: function() {
    render = function() {return null;};
  }
};

function render() {
  currentTime = new Date().getTime();
  delta = currentTime - oldTime;
  oldTime = currentTime;
  //multiplier = delta/16.6;
  multiplier = 1;

  if(!game.paused) {
    //First update all of the X and Y positions
    game.levels[currentLevel].update(multiplier);
    Player.updateAll(multiplier);

  }

    //Clear The Last Frame
    c.clearRect(0, 0, 40*u, 20*u);

    //Then Draw everything
    game.levels[currentLevel].draw();
    Player.drawAll();
    HUDElements.drawAll();


  if(game.debug) {
    debug();
  }
  //Call the next frame
  requestAnimationFrame(render);
}

console.log("Game.js Loaded");
```

index.html (default)

```
<html>
  <head>
    <title>CAGE Platformer</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
```

```
    <canvas id="gameCanvas"></canvas>

    <script src="main.js"></script>
    <script src="game.js"></script>

    <script src="modules/images.js"></script>
    <script src="modules/input.js"></script>
    <script src="modules/level.js"></script>
    <script src="modules/box.js"></script>
    <script src="modules/projectile.js"></script>
    <script src="modules/player.js"></script>
    <script src="modules/platform.js"></script>
    <script src="modules/hud.js"></script>
    <script src="modules/sprites.js"></script>
    <script src="modules/menus.js"></script>

    <script src="levels/basic.js"></script>

    <div id="splash" class="menu">
      <h1>CAGE Basic Game</h1>
      A and D to move, Space to jump, Esc to pause<br>
      <button>Press to start</button>
    </div>
    <div id="pause" class="menu">
      <h1>Game Paused</h1>
      Press Esc to Resume
    </div>

  </body>
</html>
```

index.html (platformer)

```
<html>
  <head>
    <title>CAGE Platformer</title>
    <link rel="stylesheet" href="style.css" />
  </head>
  <body>
    <canvas id="gameCanvas"></canvas>

    <script src="main.js"></script>
    <script src="modules/hud.js"></script>
    <script src="game.js"></script>
```

```
    <script src="modules/images.js"></script>
    <script src="modules/input.js"></script>
    <script src="modules/level.js"></script>
    <script src="modules/box.js"></script>
    <script src="modules/projectile.js"></script>
    <script src="modules/player.js"></script>
    <script src="modules/platform.js"></script>

    <script src="modules/mob.js"></script>
    <script src="modules/ai.js"></script>

    <script src="modules/movingPlatform.js"></script>
    <script src="modules/bouncingPlatform.js"></script>
    <script src="modules/ice.js"></script>

    <script src="modules/collectible.js"></script>
    <script src="modules/coin.js"></script>
    <script src="modules/sizePowerUp.js"></script>

    <script src="modules/crates.js"></script>

    <script src="modules/switches.js"></script>
    <script src="modules/doors.js"></script>

    <script src="modules/hudElements.js"></script>
    <script src="modules/sprites.js"></script>

    <script src="modules/menus.js"></script>

    <script src="levels/w1l1.js"></script>
    <script src="levels/w1l2.js"></script>
    <script src="levels/w1l3.js"></script>
    <script src="levels/w1boss.js"></script>

    <div id="splash" class="menu">
      <h1>CAGE Demo Game</h1>
      A and D to move, Space to jump, Esc to pause<br>
      <button>Press to start</button>
    </div>
    <div id="menu" class="menu">
      <img src="assets/chars.png"/></br>
      <button>Play as character 1</button>
      <button>Play as character 2</button>
    </div>
    <div id="pause" class="menu">
```

```
      <h1>Game Paused</h1>
      Press Esc to Resume
    </div>
    <div id="debug">
      X: <span id="x"></span><br>
      Y: <span id="y"></span><br>
      vX: <span id="vx"></span><br>
      vY: <span id="vy"></span><br>
      isJumping: <span id="ij"></span><br>
      Level Offset: <span id="lo"></span><br>
      Level X: <span id="lx"></span><br>
      FPS: <span id="fp"></span><br>
    </div>
    <script src="debug.js"></script>
  </body>
</html>
```

main.js (platformer)

```
//Key Game Variables - see document for definitions
var canvas,c,
tilesX = 40,
tilesY = 20,
pixelsPerTile = 100,
u = pixelsPerTile,
currentTime = 0,
oldTime = 0,
delta = 0,
keys = [],
modules = [],
players = [],
totalModules,
currentLevel = 0,
totalLevels,
normalGravity = 0.01*u,
BoxFriction = 0.9,
IceFriction = 0.97,
gravity = normalGravity;

//Function to initialize game. Finds the canvas, sets its size, and
calls the render loop, if levels exist to render.
window.onload = function() {
  console.log("The app has loaded.");
  reset();
```

```
  canvas = document.getElementById("gameCanvas");
  c = canvas.getContext("2d");
  canvas.height = tilesY * u;
  canvas.width = tilesX * u;

  c.imageSmoothingEnabled = false;

  totalModules = scriptCount("modules");
  totalLevels = scriptCount("levels");

  //If there are level script files loaded, AND if they are added to the
levels array correctly, start the game.
  if((totalLevels > 0) && (levels.length > 0)) {
    render();
  }
};

//Reset sets the size and position of the Chrome Window.
function reset(){
  chrome.app.window.getAll()[0].outerBounds.width  = screen.availWidth;
  chrome.app.window.getAll()[0].outerBounds.height = screen.availHeight;
  chrome.app.window.getAll()[0].outerBounds.left =  0;
  chrome.app.window.getAll()[0].outerBounds.top =  0;
}

//Loops through the included <script> tages and returns all that are in
a folder `type`
function scriptCount(type) {
  var len = 0;
  var scripts = document.getElementsByTagName("script");

  for(var i = 0; i < scripts.length;i++) {
    if(scripts[i].src.split("/")[3] == type) {
      len++;
    }
  }

  return len;
}

console.log("Main.js Loaded");
```

manifest.json

```
{
  "manifest_version": 2,
  "name": "Platformer",
  "short_name": "Platformer",
  "description": "",
  "version": "0.0.1",
  "minimum_chrome_version": "38",
  "icons": {
    "16": "assets/icon_16.png",
    "128": "assets/icon_128.png"
  },
  "app": {
    "background": {
      "scripts": ["background.js"]
    }
  },
  "permissions": ["storage"]
}
```

style.css

```
body {
    height:100%;
    width:100%;
    background-color:#589fd8;
    padding:0;
    margin:0;
    font-family:Verdana;
}

canvas {
    position:fixed;
    left:0px;
    right:0px;
    top:0px;
    bottom:0px;
    width:100%;
    margin:auto;
    z-index:-1;
    border:1px solid black;
}

#debug {
```

```
  z-index:0;
  bottom:0;
  right:0;
  background-color:rgba(0,0,0,0.5);
  color:white;
  position:fixed;
  display:none /*inline-block*/;
  width:300px;
  font-size:1.5em;
}

.menu {
  height:100%;
  width:100%;
  position:fixed;
  left:0;
  right:0;
  top:0;
  bottom:0;
  display:block;
  background-color:#444;
  color:#ffffff;
  text-align:center;
  font-family:Calibri;

}

.menu button {
  all:initial;
  height:50px;
  width:150px;
  background-color:#337733;
  color:#ffffff;
  text-align:center;
  font-family:Calibri;
  line-height:50px;
  margin:50px;
}

#splash {
  z-index:2;
  font-size:-webkit-xxx-large;
  padding-top:20vh;
}
#menu {
```

```
  z-index:1;
}
#pause {
  z-index:0;
  opacity:0.9;
  font-size:-webkit-xxx-large;
  padding-top:25vh;
  display:none;
}
```

Modules:

ai.js

```
modules.push("ai");

function NoAI() {
  return;
}

function Static() {
  this.vX = 0;
  this.vY = 0;
}

function LinearAI() {
  this.vX = this.aiData.vX;
}

function PatrolAI() {
  if(this.aiData.dir == 1) {
    this.vX = this.aiData.vX;
  }
  if(this.aiData.dir == -1) {
    this.vX = -this.aiData.vX;
  }
  var offset = game.levels[currentLevel].offset;
  if(this.x+offset <= this.aiData.x1*u) {
    this.aiData.dir = 1;
  }
  if(this.x+offset >= (this.aiData.x2*u) - this.w) {
    this.aiData.dir = -1;
  }
}

function VerticalPatrolAI() {
```

```
  this.vY = this.aiData.dir * this.aiData.vY;
  this.vY -= gravity - 1;

  if(this.y <= this.aiData.y1*u) {
    this.aiData.dir = 1;
  }
  if(this.y >= (this.aiData.y2*u)) {
    this.aiData.dir = -1;
  }
}

function Boss1AI() {
  //drops the door into the arena to stop the player from leaving
  if(!this.locked) {
    doorArray[0].close();
    this.aiData.locked = true;
  }
  //Make sure the Player doesn't leave the arena
  if(Player.getLeader().x < 0) {
    Player.getLeader().x = 0;
  }

  //Make sure the Boss can't jump indefinitely
  if(!this.aiData.isJumping) {
    this.aiData.isJumping = false;
  }

  //Locks the levels scrolling once the player is within the arena
  if(game.levels[currentLevel].offset == 4000) {
    w1boss.scrollLock = true;
  }

  //controls boss death
  if(this.aiData.health === 0) {
    doorArray[0].open();
    doorArray[1].open();
    w1boss.scrollLock = false;
    this.kill();
    bossPlat.aiData.x1 = 41.05;
    bossPlat.aiData.x2 = 79.95;
  }

  //Initialises the boss battle once the door is locked
  if(this.aiData.stage === undefined && this.aiData.locked) {
    bossMob.aiData.stage = 1;
```

```
    if(w1boss.dynamics.length > 2) {
      w1boss.dynamics.length = 2;
    }
    setTimeout(function(){
      bossMob.aiData.enemies = 1;
      bossMob.aiData.dead = 0;
    },2000);
  }
  //Stage 1 of the fight: Spawn a patrol mob
  if((this.aiData.stage === 1) && (this.aiData.enemies === 1)) {
    game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:8,dir:-1},function(){
bossMob.aiData.dead++; this.kill();}));
    this.aiData.enemies = 0;
  }

  //Once the first patrol mob is killed, advance the boss fight
  if((this.aiData.stage === 1) && (this.aiData.dead === 1)) {
    this.aiData.enemies = 2;
    this.aiData.stage = 2;
  }

  //Spawn two, faster patrol mobs 1 second apart
  if((this.aiData.stage === 2) && (this.aiData.enemies > 0)) {
    setTimeout(function(){
      game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:15,dir:-1},function()
{bossMob.aiData.dead++;this.kill();}));
      setTimeout(function(){
        game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:15,dir:-1},function()
{bossMob.aiData.dead++;this.kill();}));
      },1000);
    },1000);
    this.aiData.enemies = 0;
  }

  //Once both of these enemies are dead, advance the boss battle
  if((this.aiData.stage === 2) && (this.aiData.dead === 3)) {
    this.aiData.enemies = 4;
    this.aiData.stage = 3;
  }

  //Spawn 4 faster patrol mobs 1 second apart
  if((this.aiData.stage === 3) && (this.aiData.enemies > 0)) {
```

```
    setTimeout(function(){
      game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},function()
{bossMob.aiData.dead++;this.kill();}));
      setTimeout(function(){
        game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},function()
{bossMob.aiData.dead++;this.kill();}));
        setTimeout(function(){
          game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},function()
{bossMob.aiData.dead++;this.kill();}));
          setTimeout(function(){
            game.levels[currentLevel].dynamics.push(new
Mob(32,16,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},function()
{bossMob.aiData.dead++;this.kill();}));
          },1000);
        },1000);
      },1000);
    },1000);
    this.aiData.enemies = 0;
  }
  if((this.aiData.stage === 3) && (this.aiData.dead === 7)) {
    //Advance the mob stage and make the movingPlatform start moving
    this.aiData.enemies = 8;
    this.aiData.stage = 4;
    bossPlat.aiData.vX = 4;
  }
  if((this.aiData.stage === 4) && (this.aiData.enemies > 0)) {
    setTimeout(function(){
      //Pushes 8 new patrol mobs to the level, 750 ms apart
      game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},f
unction(){bossMob.aiData.dead++;this.kill();}));
      setTimeout(function(){
        game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:1},fu
nction(){bossMob.aiData.dead++;this.kill();}));
        setTimeout(function(){
          game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},f
unction(){bossMob.aiData.dead++;this.kill();}));
          setTimeout(function(){
            game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:1},fu
```

```
nction(){bossMob.aiData.dead++;this.kill();}));
            setTimeout(function(){
              game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},f
unction(){bossMob.aiData.dead++;this.kill();}));
              setTimeout(function(){
                game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:1},fu
nction(){bossMob.aiData.dead++;this.kill();}));
                setTimeout(function(){
                  game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:-1},f
unction(){bossMob.aiData.dead++;this.kill();}));
                  setTimeout(function(){
                    game.levels[currentLevel].dynamics.push(new
Mob(bossMob.x/u,14,1,1,PatrolAI,{dmg:5,x1:40.05,x2:73.95,vX:20,dir:1},fu
nction(){bossMob.aiData.dead++;this.kill();}));
                  },750);
                },750);
              },750);
            },750);
          },750);
        },750);
      },2500);
      this.aiData.enemies = 0;
    }
    //Once all the mobs are dead, return the platform to the right and
    //advance the boss battle
    if((this.aiData.stage === 4) && (this.aiData.dead === 15)) {
      this.aiData.enemies = 1;
      this.aiData.stage = 5;
      bossPlat.aiData.dir = 1;
    }

    //Once the platform is at its original position, lift it up, and make
    //it patrol over the arena.
    if(this.aiData.stage == 5) {
      if(bossPlat.x > 3199) {
        bossPlat.aiData.vX = 0;
        bossPlat.vY = -8;
        bossMob.x -= 4;
        if(bossPlat.y <= 600) {
          bossPlat.vY = 0;
          bossPlat.y = 600;
```

```
        bossPlat.aiData.vX = 4;
        this.aiData.stage = 6;
        this.aiData.x1 = 41.1;
        this.aiData.x2 = 71.95;
        this.aiData.vX = 10;
        this.aiData.dir = 1;
      }
    }
  }

  //Make the boss jump off the platform, and patrol the arena, jumping
  //randomly.
  if(this.aiData.stage == 6) {
    if(bossMob.y < 1000) {
      bossMob.x -= 4;
    }
    if((Math.random() < 0.01) && (!this.aiData.isJumping)) {
      bossMob.vY = -25;
      this.aiData.isJumping = true;
    }
    //Patrol AI code. function cannot be used as context of "this" has
changed
    if(this.aiData.dir == 1) {
      this.vX = this.aiData.vX;
    }
    if(this.aiData.dir == -1) {
      this.vX = -this.aiData.vX;
    }
    var offset = game.levels[currentLevel].offset;
    if(this.x+offset <= this.aiData.x1*u) {
      this.aiData.dir = 1;
    }
    if(this.x+offset >= (this.aiData.x2*u) - this.w) {
      this.aiData.dir = -1;
    }
  }
}

function Boss1Hit() {
  this.aiData.health -= 1;
}
```

bouncingPlatform.js

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module
bouncingPlatform";
} else {
  //Push "platform" to list of included modules
  modules.push("box");

  var BouncingPlatform = class BouncingPlatform extends Box {
    constructor(x,y,w,h,tile) {
      super(x,y,w,h,false);
      this.t = tile;
      this.vX = 0;
      this.vY = 0;

    }

    draw() {
      this.t.draw(this.x,this.y,this.w,this.h);
    }

    update() {}

    reset() {}

    move() {}

    collision(obj,dir) {
      if(dir == "b") {
        obj.vY = -0.9*Math.abs(obj.vY);
        if(obj.vY < -obj.vYmax) {
          obj.vY = -obj.vYmax;
        }
        if(Math.abs(obj.vY) < 1) {
          obj.vY = 0;
        }
        obj.vY -= gravity;
      }
    }
  };

  BouncingPlatform.type = "dynamic";
}
```

box.js

```
//Depends upon module "level"
if(modules.indexOf("level") == -1) {
  throw "DependancyError: Module level is required for Module box";
} else {
  //Push "box" to the list of modules included
  modules.push("box");

  //Define the box Class
  var Box = class Box {
    constructor(x, y, width, height, solid) {
      this.x = x*u;
      this.y = y*u;
      this.h = height*u;
      this.w = width*u;
      this.s = solid;
    }

    draw() {
      c.fillStyle = "green";
      c.fillRect(this.x,this.y,this.w,this.h);
    }

    //Collision algorithm from somethinghitme.com
    static colCheck(a,b) {
      // get the vectors to check against
      var vX = (a.x + (a.w/ 2)) - (b.x + (b.w / 2)),
          vY = (a.y + (a.h / 2)) - (b.y + (b.h / 2)),
          // add the half widths and half heights of the objects
          hWidths = (a.w / 2) + (b.w / 2),
          hHeights = (a.h / 2) + (b.h / 2),
          colDir = null;
// if the x and y vector are less than the half width or half height,
// they we must be inside the object, causing a collision
      if (Math.abs(vX) < hWidths && Math.abs(vY) < hHeights) {
          // figures out on which side we are colliding
          //(top, bottom, left, or right)
          var oX = hWidths - Math.abs(vX),
              oY = hHeights - Math.abs(vY);
          if (oX >= oY) {
              if (vY > 0) {
                  colDir = "t";
          //If the second item is solid, move the first outside of it.
                  if (b.s) {
                      a.y += oY;
```

```
                    a.vY = 0;
                }
            } else {
                colDir = "b";
                if (b.s) {
                    a.y -= oY;
                    a.vY = 0;
                }
            }
        } else {
            if (vX > 0) {
                colDir = "l";
                if (b.s) {
                    a.x += oX;
                    a.vX = 0;
                }
            } else {
                colDir = "r";
                if (b.s) {
                    a.x -= oX;
                    a.vX = 0;
                }
            }
        }
    }
    return colDir;
  }
};

Box.type = "static";
}

console.log("Box.js Loaded");
```

breakablePlatform.js

```
//Depends on module "platform", "platform", “level”
if(modules.indexOf("platform") == -1) {
  throw "DependancyError: Module platform is required for Module
breakablePlatform";
} else if(modules.indexOf("player") == -1) {
  throw "DependancyError: Module player is required for Module
breakablePlatform";
} else if(modules.indexOf("level") == -1) {
  throw "DependancyError: Module level is required for Module
```

```
breakablePlatform";
} else {
  //Push "platform" to list of included modules
  modules.push("breakablePlatform");

  var BreakablePlatform = class BreakablePlatform extends Platform {
    constructor(x,y,w,h,tile,data) {
      super(x,y,w,h,tile);
      this.d = data;
    }

    draw() {
      this.t.draw(this.x,this.y,this.w,this.h);
    }

    update() {}

    reset() {
        if(this.destroyed) {
            this.destroyed = false;
        }
    }

    move() {}

    collision(obj,dir) {
      if(obj.constructor.name == "Player") {
        if(dir == "t") {
            if(this.d.item) {
            game.levels[currentLevel].collectibles.push(this.d.item);
          }
        }
      }
    }
  };

  BreakablePlatform.type = "dynamic";
}
```

coin.js

```
if(modules.indexOf("collectible") == -1) {
  throw "DependancyError: Module collectible is required for Module
coin";
} else {
```

```
  modules.push("coin");

  var Coin = class Coin extends Collectible {
    constructor(x,y,t) {
      super(x,y,0.3,0.3,t);
    }

    collect(x) {
      if(!this.collected) {
        var collectedBy = x;
        if(!collectedBy.coins) {
          collectedBy.coins = 0;
        }
        collectedBy.coins++;
        this.collected = true;
      }
    }
  };

  Coin.type = "collectible";
}
```

collectible.js

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module
collectible";
} else {
  modules.push("collectible");

  var Collectible = class Collectible extends Box {
    constructor(x,y,w,h,tile) {
      super(x,y,w,h,false);
      this.collected = false;
      this.t = tile;
    }

    update() {
      var i = players.length;
      var dir = "";
      while(i--) {
        dir = Box.colCheck(this, players[i]);
        if(dir !== null) {
          this.collect(players[i]);
```

```
        }
      }
    }

    collect() {
      this.collected = true;
    }

    draw() {
      if (!this.collected) {
        this.t.draw(this.x,this.y,this.w,this.h);
      }
    }
  };

  Collectible.type = "collectible";
}
```

crates.js

```
//Depends on module "box"
if(modules.indexOf("projectile") == -1) {
  throw "DependancyError: Module projectile is required for Module
crate";
} else {
  //Push "platform" to list of included modules
  modules.push("crates");

  var Crate = class Crate extends Projectile {
    constructor(x,y,h,w,resettable,contents) {
      super(x,y,h,w,0,0,false);
      this.c = contents;
      this.friction = BoxFriction;
      this.isOpened = false;
      this.resettable = resettable;
      this.initial = {
        x: x*u,
        y: y*u
      };
    }

    draw() {
      if(!this.isOpened) {
        c.fillStyle = "yellow";
        c.fillRect(this.x,this.y,this.w,this.h);
      }
```

```
  }

  update(multiplier) {
    if(!this.isOpened) {
      this.move(multiplier);
      this.c.x = this.x;
      this.c.y = this.y;
      this.vY += gravity;
      this.vX *= Math.pow(this.friction,multiplier);
      var dir = game.levels[currentLevel].colCheck(this);
      this.checkDir(dir);
    }
  }

  collision(obj,dir) {
    if(!this.isOpened) {
      if(obj.constructor.name == "Player") {
        if(keys[obj.controls.open]) {
          this.destroy();
        }

        if ((dir == "l") || (dir == "r")){
          this.vX = obj.vX;
        } else if(dir == "b") {
          obj.y = this.y - obj.h;
          obj.vY = -gravity;
        }
      }
    }
  }

  destroy() {
    if(!this.isOpened) {
      game.levels[currentLevel].collectibles.push(this.c);
      this.isOpened = true;
    }
  }

  reset() {
    if(this.resettable) {
      this.x = this.initial.x;
      this.y = this.initial.y;
      this.isOpened = false;
    }
  }
```

```
    checkDir(dir) {
      if(dir.indexOf("Ice") != -1) {
        this.friction = IceFriction;
      } else {
        this.friction = BoxFriction;
      }
    }
  };

  Crate.type = "dynamic";
}
```

doors.js

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module switch";
} else {
  //Push "platform" to list of included modules
  modules.push("switch");

  var doorArray = [];
  var Door = class Door extends Platform {
    constructor(x,y,w,h,tile,defaultPosition,id) {
      super(x,y,w,h,tile);
      this.isOpen = defaultPosition;
      this.s = !defaultPosition;
      this.defaultPosition = defaultPosition;
      this.id = id;
      doorArray[id] = this;
    }

    draw() {
      if(!this.isOpen) {
        this.t.draw(this.x,this.y,this.w,this.h);
      }
    }

    update(){
      if(this.isOpen){
        this.close();
      } else {
        this.open();
      }
    }
```

```
    reset() {
      this.isOpen = this.defaultPosition;
      this.s = true;
    }

    open() {
      this.isOpen = true;
      this.s = false;
    }

    close() {
      this.isOpen = false;
      this.s = true;
    }
  };
  Door.type = "static"; //Stops door being tested 60x per second.
door.update() is called when switch is fired.
}
```

hud.js

```
modules.push("HUD");
var HUDElements = class HUDElements {
  static drawAll() {
    var i = game.hudElements.length;
    while(i--) {
      game.hudElements[i].draw();
    }
  }
};

var HUDBar = class HUDBar {
  constructor(x,y,w,h,bgcol,textcol,max,getValue) {
    this.x = x*u;
    this.y = y*u;
    this.w = w*u;
    this.h = h*u;
    this.bgCol = bgcol;
    this.textCol = textcol;
    this.maxVal = max;
    this.getVal = getValue;
    game.hudElements.push(this);
  }

  draw() {
```

```
    var num = null;
    if(!game.paused) {
      num = this.getVal();
    }
    if(num !== null) {
      var width = (num/this.maxVal)*this.w;
      c.fillStyle = this.bgCol;
      c.strokeRect(this.x,this.y,this.w,this.h);
      c.fillRect(this.x,this.y,width,this.h);
      c.font = '5em Calibri';
      c.fillStyle = this.textCol;
      c.fillText(num.toString(), this.x+10, this.y+65);
    }
  }
};
```

hudElements.js (platformer)

```
new HUDBar(33,1,6,1,"#FF0000","#FFFFFF",100,() => {return
game.players[0].health;});
```

ice.js

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module ice";
} else {
  modules.push("ice");

  var Ice = class Ice extends Box {
    draw() {
      this.t.draw(this.x,this.y,this.w,this.h);
    }
  };
  Ice.type = "static";
}
```

images.js

```
modules.push("image");

var BlockColTile = class BlockColourTile {
  constructor(color) {
    this.col = color;
  }
```

```
  draw(x,y,w,h) {
    c.fillStyle = this.col;
    c.fillRect(x,y,w,h);
  }
};

var BlockColSprite = class BlockColourSprite {
  constructor(color) {
    this.col = color;
  }

  draw(obj) {
    c.fillStyle = this.col;
    c.fillRect(obj.x,obj.y,obj.w,obj.h);
  }
};

var Tile = class Tile {
  constructor(url,cx,cy,cw,ch) {
    this.img = new Image();
    this.img.src = url;
    this.cx = cx;
    this.cy = cy;
    this.cw = cw;
    this.ch = ch;
  }

  draw(x,y,w,h) {
    c.drawImage(this.img,this.cx,this.cy,this.cw,this.ch,x,y,w,h);
  }
};

//Draws an image at a fixed location in the level
var Sprite = class Sprite extends Tile {
  constructor(url,cx,cy,cw,ch,x,y,w,h, offsetFactor) {
    super(url,cx,cy,cw,ch);
    this.x = x;
    this.y = y;
    this.w = w;
    this.h = h;
    this.offsetFactor = offsetFactor;
  }

  draw(offset) {
    var drawOffset = offset * this.offsetFactor;
```

```
c.drawImage(this.img,this.cx,this.cy,this.cw,this.ch,this.x-drawOffset,t
his.y,this.w,this.h);
  }
};

var SpriteSet = class SpriteSet {
  constructor(images) {
    //Images is an array of background layers. images[0] is drawn at the
top
    this.layers = images;
    this.layerCount = images.length;
  }
};

var Background = class Background extends SpriteSet {
  draw(offset) {
    var x = this.layerCount;
    while(x--) {
      this.layers[x].draw(offset);
    }
  }
};

var RepeatingTile = class RepeatingTile extends Tile {
  constructor(url) {
    super(url,0,0,0,0);
  }
  draw(x,y,w,h) {
    c.translate(x,y);
    var style = c.createPattern(this.img,"repeat");
    c.fillStyle = style;
    c.fillRect(0,0,w,h);
    c.translate(-x,-y);
  }
};

var noBg = {draw:function(){}};
var noFg = noBg;
var noTile = noBg;
```

input.js

```
modules.push("input");

window.addEventListener("load",function() {
```

```
  canvas.addEventListener("mousedown", CanvasClickHandler, false);
  canvas.style.zIndex = 1;
});

function CanvasClickHandler(e) {

}

function getPosition(event) {
  evX = event.x;
  evY = event.y;
  evX -= canvas.offsetLeft;
  evY -= canvas.offsetTop;
  return {
    canvasX: evX,
    canvasY: evY,
    screenX: event.x,
    screenY: event.y,
    gameX: tilesX * u * evX/window.innerWidth,
    gameY: tilesY * u * evY/window.innerHeight
  };
}

document.body.addEventListener("keydown", function(e) {
  if ((!keys[27]) && (e.keyCode == 27)) {
    if(!game.paused) {
      game.pause();
      pause.style.display = "block";
    } else {
      game.play();
      pause.style.display = "none";
    }
  }
  keys[e.keyCode] = true;
});
document.body.addEventListener("keyup", function(e) {
  keys[e.keyCode] = false;
});

console.log("Input.js Loaded");
```

level.js

```
modules.push("level");

var Level = class Level {
```

```
  constructor(length,background,foreground) {
    this.len = length*u;
    this.offset = 0;
    this.statics = [];
    this.dynamics = [];
    this.collectibles = [];
    this.background = background;
    this.foreground = foreground;
    this.scrollLock = false;
    game.levels.push(this);
    this.index = game.levels.length-1;
  }

  add(...args) {
    for(var i in args) {
      var obj = args[i];
      if(obj.constructor.type == "static") {
        this.statics.push(obj);
      } else if(obj.constructor.type == "dynamic") {
        this.dynamics.push(obj);
      } else if(obj.constructor.type == "collectible") {
        this.collectibles.push(obj);
      } else {
        console.error("Error adding to scene: object",obj,"is not of a
valid type");
      }
    }
  }

  //Loop through every item in the scene and see if obj has collided
with it,
  //Returns an array of all the collision directions relative to obj.
  colCheck(obj) {
    var dir = [];
    var temp = "";
    var i = this.statics.length;
    while(i--) {
      temp = Box.colCheck(obj,this.statics[i]);
      if(temp !== null) {
        dir.push(temp);
        dir.push(this.statics[i].constructor.name);
      }
    }
    i = this.dynamics.length;
    while(i--) {
```

```
      if(this.dynamics[i] != obj) {
        temp = Box.colCheck(obj,this.dynamics[i]);
        if(temp !== null) {
          dir.push(temp);
          dir.push(this.dynamics[i].constructor.name);
          this.dynamics[i].collision(obj,temp);
        }
      }
    }
    return dir;
  }

  update(multiplier) {
    //Update all the entities that may have changed
    var i = this.dynamics.length;
    while(i--) {
      this.dynamics[i].update(multiplier);
    }

    i = this.collectibles.length;
    while(i--) {
      this.collectibles[i].update();
    }
    //Get the rightmost player
    var lead = Player.getLeader();

    //If they are in the center of the screen and not at the end of the
level, scroll the level left
    if((lead.vX > 0.1) && (lead.x > 18*u) && (this.offset < (this.len -
(40*u)) && !this.scrollLock)) {
      this.scroll(lead.vX);
    }
    //If they are in the center of the screen and not at the start of
the level, scroll the level right
    if((lead.vX < -0.1) && (lead.x < 17*u) && (this.offset > 0) &&
!this.scrollLock) {
      this.scroll(lead.vX);
    }

    //Make sure the level is not scrolled past its max and minimum
    if(this.offset < 0) {
      this.scroll(-this.offset);
    }
    if(this.offset > this.len - 40*u) {
      this.scroll((this.len - (40*u)) - this.offset);
```

```
    }

    //Handle level switching
    if(lead.x > 40*u) {
      Player.moveAll(1,10);
      currentLevel++;
    }
    if((lead.x < 0) && (currentLevel > 0)) {
      Player.moveAll(39,10);
      currentLevel--;
    }
    if(currentLevel >= game.levels.length) {
      currentLevel = 0;
    }
  }

  draw() {
    this.background.draw(this.offset);
    var i = this.statics.length;
    while(i--) {
      this.statics[i].draw();
    }

    i = this.dynamics.length;
    while(i--) {
      this.dynamics[i].draw();
    }

    i = this.collectibles.length;
    while(i--) {
      this.collectibles[i].draw();
    }

    this.foreground.draw(this.offset);
  }

  scroll(x) {
    var i = this.dynamics.length;
    while(i--) {
      this.dynamics[i].x -= x;
    }
    i = this.statics.length;
    while(i--) {
      this.statics[i].x -= x;
    }
```

```
    i = this.collectibles.length;
    while(i--) {
      this.collectibles[i].x -= x;
    }
    this.offset += x;
    Player.scroll(x);
  }

  reset() {
    this.scroll(-this.offset);
    var i = this.dynamics.length;
    while(i--) {
      this.dynamics[i].reset();
    }
  }
};


console.log("Level.js Loaded");
```

menus.js (platformer)

```
window.addEventListener("load",function() {
  document.querySelector("#menu button:first-of-type").onclick =
function() {
    new Player(1,1,{left:65,right:68,up:32,open:83},AlexSprite);
    game.play();
    document.getElementById('menu').style.display='none';
  };
  document.querySelector("#menu button:last-of-type").onclick =
function() {
    new Player(1,1,{left:65,right:68,up:32,open:83},MaxSprite);
    game.play();
    document.getElementById('menu').style.display='none';
  };
  document.querySelector("#splash button:first-of-type").onclick =
function() {
    document.getElementById('splash').style.display='none';
  };
});window.addEventListener("load",function() {
  document.querySelector("#menu button:first-of-type").onclick =
function() {
    new Player(1,1,{left:65,right:68,up:32,open:83},AlexSprite);
    game.play();
    document.getElementById('menu').style.display='none';
```

```
  };
  document.querySelector("#menu button:last-of-type").onclick =
function() {
    new Player(1,1,{left:65,right:68,up:32,open:83},MaxSprite);
    game.play();
    document.getElementById('menu').style.display='none';
  };
  document.querySelector("#splash button:first-of-type").onclick =
function() {
    document.getElementById('splash').style.display='none';
  };
});
```

mob.js

```
//Depends on modules projectile
if(modules.indexOf("projectile") == -1) {
  throw "Dependancy Error: Module projectile is required for Module
player";
} else {
  modules.push("mob");

  var Mob = class Mob extends Projectile {
    constructor(x,y,h,w,ai,aiData,aiHit) {
      super(x,y,h,w,1,1,true);
      this.initial = {
        x: x*u,
        y: y*u
      };
      this.ai = ai;
      this.aiData = aiData;
      this.aiHit = aiHit;
      if(aiHit === undefined) {
        this.aiHit = this.kill;
      }
      this.encountered = false;
      this.dead = false;
      this.friction = BoxFriction;
    }

    update() {
      if(!this.dead) {
        if((!this.encountered) && (this.x < 40*u)) {
          this.encountered = true;
        }
        if(this.encountered) {
```

```
        this.ai();
        this.move(multiplier);
        var dir = game.levels[currentLevel].colCheck(this);
        this.checkDir(dir);
        this.vX *= Math.pow(this.friction,multiplier);
        this.vY += gravity * multiplier;
      }
    }
  }

  kill() {
    this.dead = true;
    this.x = 50*u;
    this.y = 50*u;
  }


  reset() {
    this.dead = false;
    this.encountered = false;
    this.x = this.initial.x;
    this.y = this.initial.y;
  }

  collision(obj,dir) {
    //Dynamic Object has collided with mob
    var objType = obj.constructor.name;
    if(objType == "Player") {
      if(dir == "b") {
        this.aiHit();
        obj.vY = -1*obj.vYmax;
        obj.isJumping = false;
      } else {
        obj.hit(this.aiData.dmg);
        if(dir == "l") {
          obj.vY = -obj.vYmax;
          obj.vX = obj.vXmax;
          obj.isJumping = true;
        }
        if(dir == "r") {
          obj.vY = -obj.vYmax;
          obj.vX = -obj.vXmax;
          obj.isJumping = true;
        }
      }
```

```
      }
      if(objType == "Mob") {
        if(dir == "r") {
          this.x -= obj.w;
          obj.x += this.w ;
        }
        if(dir == "l") {
          this.x += obj.w;
          obj.x -= this.w ;
        }
      }
    }

    checkDir(dir) {
      if(dir.indexOf("b") != -1) {
        this.aiData.isJumping = false;
      }
      if(dir.indexOf("Ice") != -1) {
        this.friction = IceFriction;
      } else {
        this.friction = BoxFriction;
      }
    }

    draw() {
      if ((this.encountered) && (!this.dead)) {
        MobSprite.draw(this);
      }
    }
  };
  Mob.type = "dynamic";
  console.log("Mob.js loaded");
}
```

movingPlatform.js

```
//Depends on module "ai", "platform"
if(modules.indexOf("platform") == -1) {
  throw "DependancyError: Module platform is required for Module
movingPlatform";
} else if(modules.indexOf("ai") == -1) {
  throw "DependancyError: Module ai is required for Module
movingPlatform";
} else {
```

```
//Push "platform" to list of included modules
modules.push("movingPlatform");

var MovingPlatform = class MovingPlatform extends Platform {
  constructor(x,y,w,h,tile,ai,aiData) {
    super(x,y,w,h,tile);
    this.vX = 0;
    this.vY = 0;
    this.encountered = false;
    this.ai = ai;
    this.aiData = aiData;
    this.initial = {
      x: x*u,
      y: y*u
    };
  }

  draw() {
    this.t.draw(this.x,this.y,this.w,this.h);
  }

  update() {
    if((!this.encountered) && (this.x < 40*u)) {
      this.encountered = true;
    }
    if(this.encountered) {
      this.ai();
      this.move();
    }
  }

  reset() {
    this.x = this.initial.x;
    this.y = this.initial.y;
  }

  move() {
    this.x += this.vX;
    this.y += this.vY;
  }

  collision(obj,dir) {
    if(dir == "b") {
      obj.x += this.vX;
      obj.y += this.vY;
```

```
        if((obj == Player.getLeader()) &&
(game.levels[currentLevel].offset < (game.levels[currentLevel].len -
(40*u)))) {
          game.levels[currentLevel].scroll(this.vX);
        }
      }
    }
  };
  MovingPlatform.type = "dynamic";
}
```

platform.js

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module platform";
} else {
  //Push "platform" to list of included modules
  modules.push("platform");

  var Platform = class Platform extends Box {
    constructor(x,y,w,h,tile) {
      super(x,y,w,h,true);
      this.t = tile;
    }

    draw() {
      this.t.draw(this.x,this.y,this.w,this.h);
    }
  };
  Platform.type = "static";
}
```

player.js

```
//Depends on modules projectile and input
if(modules.indexOf("projectile") == -1) {
  throw "Dependancy Error: Module projectile is required for Module
player";
} else if(modules.indexOf("input") == -1) {
  throw "Dependancy Error: Module input is required for Module player";
} else {
  //Push "player" to the list of included modules.
  modules.push("player");

  //Player is a projectile with max XY velocity, friction and
input-based velocity
```

```
Player = class Player extends Projectile {
  constructor(x,y,controls,sprite) {
    super(x,y,1,2,10,0,true);
    this.controls = controls;
    this.vXmax = (1/6)*u;
    this.vYmax = (1/4)*u;
    this.friction = BoxFriction;
    this.isJumping = false;
    this.sprite = sprite;
    this.health = 100;
    this.playerNumber = game.players.length;
    game.players.push(this);
  }

  static updateAll(multiplier) {
    var i = game.players.length;
    while(i--){
      game.players[i].update(multiplier);
    }
  }

  static drawAll() {
    var i = game.players.length;
    while(i--){
      game.players[i].draw();
    }
  }

  static moveAll(x,y) {
    var i = game.players.length;
    while(i--){
      game.players[i].x = x*u;
      game.players[i].y = y*u;
    }
  }


  static getLeader() {
    var i = game.players.length;
    var leader = game.players[0];
    while(i--){
      if(game.players[i].x > leader.x) {
        leader = game.players[i];
      }
    }
```

```
      return leader;
    }

    static scroll(x) {
      var i = game.players.length;
      while(i--){
        game.players[i].x -= x;
      }
    }
    //Polymorphic routine, called on every dynamic object every frame
    //Checks keys pressed, moves and checks collisions with all items in
the current level
    //Dir contains a direction, or null if there is no collision.
    update(multiplier) {
      if(this.y > 20*u) {
        this.kill();
        return;
      }
      this.checkKeys(multiplier);
      this.move(multiplier);
      var dir = game.levels[currentLevel].colCheck(this);
      this.checkDir(dir);
    }

    kill() {
      game.levels[currentLevel].reset();
      this.x = u;
      this.y = u;
      this.w = u;
      this.h = 2*u;
      this.vX = 0;
      this.vY = 0;
      this.health = 100;
    }

    draw() {
      this.sprite.draw(this);
    }

    checkKeys(multiplier) {
      if (keys[this.controls.left]) {
          // left arrow
          if (this.vX > -this.vXmax) {
            if(!this.isJumping) {
              this.vX -= this.vXmax/4;
```

```
          } else {
            this.vX -= this.vXmax* (1/8);
          }
        }
    }
    if (keys[this.controls.right]) {
      // right arrow
      if (this.vX < this.vXmax) {
          if(!this.isJumping) {
            this.vX += this.vXmax/4;
          } else {
            this.vX += this.vXmax * (1/8);
          }
      }
    }
    if (keys[this.controls.up]) {
      if (!this.isJumping && this.vY <= 0) {
        this.vY -= this.vYmax/5;
        if(this.vY < -this.vYmax) {
          this.isJumping = true;
        }
      }
    }
    this.vX *= Math.pow(this.friction,multiplier);
    this.vY += gravity * multiplier;
  }

  checkDir(dir) {
    if(dir.indexOf("b") != -1) {
      this.isJumping = false;
    }
    if(dir.indexOf("t") != -1) {
      this.vY = 0;
    }
    if(dir.indexOf("Ice") != -1) {
      this.friction = IceFriction;
    } else if(dir.indexOf("Platform") != -1) {
      this.friction = BoxFriction;
    }
  }

  hit(damage) {
    this.health -= damage;
    if(this.health <= 0) {
      this.kill();
```

```
      }
    }
  };
  Player.type = "dynamics";
}

console.log("Player.js Loaded");
```

projectile.js

```
//Depends on module "box"
if(modules.indexOf("box") == -1) {
  throw "DependancyError: Module box is required for Module projectile";
} else {
  //Push "projectile" to list of included modules
  modules.push("projectile");

  //super() calls the Box constructor, before adding vX and vY
properties, which store XY velocity
  var Projectile = class Projectile extends Box {
    constructor(x,y,height,width,vX,vY,solid) {
      super(x,y,height,width,solid);
      this.vX = vX;
      this.vY = vY;
    }

    move(multiplier) {
      this.x += this.vX * multiplier;
      this.y += this.vY * multiplier;
    }
  };
  Projectile.type = "dynamic";
}

console.log("Projectile.js Loaded");
```

sizePowerUp.js

```
if(modules.indexOf("collectible") == -1) {
  throw "DependancyError: Module collectible is required for Module
sizePowerUp";
} else {
  modules.push("sizePowerUp");

  var SizePowerUp = class SizePowerUp extends Collectible {
    constructor(x,y,tile,factor,permanent) {
      super(x,y,0.5,0.5,tile);
```

```
      this.ix = x*u;
      this.iy = y*u;
      this.factor = factor;
      this.permanent = permanent;
      this.collected = false;
    }

    collect(x) {
      if(!this.collected) {
        this.x = this.ix;
        this.y = this.iy;

        var collectedBy = x;
        if(!this.permanent) {
          this.collected = true;
        }
        collectedBy.w = this.factor * u;
        collectedBy.h = this.factor * 2 * u;
      }
    }
  };
  SizePowerUp.type = "collectible";
}
```

sprites.js (platformer)

```
//Draws player depending on players' state
var PlayerSprite = class PlayerSprite {
  constructor(img,data) {
    this.img = new Image();
    this.img.src = img;
    this.d = data;
  }

  draw(player) {
    /* sw: Sprite Width
     * sh: Sprite Height
     * drx: Falling, Facing Right,X Co-ord
     * dry: Falling, Facing Right,Y Co-ord
     * dlx: Falling, Facing Left,X Co-ord
     * dly: Falling, Facing Left,Y Co-ord
     * urx: Jumping, Facing Right,X Co-ord
     * ury: Jumping, Facing Right,Y Co-ord
     * ulx: Jumping, Facing Left,X Co-ord
     * uly: Jumping, Facing Left,Y Co-ord
     * lbx: Skidding, facing left, X Co-ord
```

```
     * lby: Skidding, facing left, Y Co-ord
     * rbx: Skidding, facing right, X Co-ord
     * rby: Skidding, facing right, Y Co-ord
     * l1x: Walking Left, Frame 1, X Co-ord
     * l1y: Walking Left, Frame 1, Y Co-ord
     * l2x: Walking Left, Frame 2, X Co-ord
     * l2y: Walking Left, Frame 2, Y Co-ord
     * r1x: Walking Right, Frame 1, X Co-ord
     * r1y: Walking Right, Frame 1, Y Co-ord
     * r2x: Walking Right, Frame 2, X Co-ord
     * r2y: Walking Right, Frame 2, Y Co-ord
     * slx: Standing, Facing Left, X Co-ord
     * sly: Standing, Facing Left, Y Co-ord
     * srx: Standing, Facing Right, X Co-ord
     * sry: Standing, Facing Right, Y Co-ord
     * frn: Frame Number
     * afn: Animation Frame Number;
    */
    if((player.vY > 1) && (player.vX >= 0)) {

c.drawImage(this.img,this.d.drx,this.d.dry,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }
    if((player.vY < -1) && (player.vX >= 0)){

c.drawImage(this.img,this.d.urx,this.d.ury,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }

    if((player.vY > 1) && (player.vX < 0)) {

c.drawImage(this.img,this.d.dlx,this.d.dly,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }
    if((player.vY < -1) && (player.vX < 0)){

c.drawImage(this.img,this.d.ulx,this.d.uly,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }

    if((player.vX > 1) && (keys[player.controls.left])) {
```

```
c.drawImage(this.img,this.d.lbx,this.d.lby,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }
    if((player.vX < -1) && (keys[player.controls.right])) {

c.drawImage(this.img,this.d.rbx,this.d.rby,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }
    if((player.vX > 1)) {
      this.d.frn += Math.floor(100/delta);
      if(this.d.frn > 40) {
        this.d.afn++;
        this.d.frn -= 40;
      }
      if(this.d.afn % 2 === 0) {

c.drawImage(this.img,this.d.r1x,this.d.r1y,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      } else {

c.drawImage(this.img,this.d.r2x,this.d.r2y,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      }
      return;
    }
    if((player.vX < -1)) {
      this.d.frn += Math.floor(100/delta);
      if(this.d.frn > 40) {
        this.d.afn++;
        this.d.frn -= 40;
      }
      if(this.d.afn % 3 === 0) {

c.drawImage(this.img,this.d.l1x,this.d.l1y,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      } else {

c.drawImage(this.img,this.d.l2x,this.d.l2y,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      }
      return;
    }
```

```
    if(player.vX >= 0) {

c.drawImage(this.img,this.d.srx,this.d.sry,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }

    if(player.vX < 0) {

c.drawImage(this.img,this.d.slx,this.d.sly,this.d.sw,this.d.sh,player.x,
player.y,player.w,player.h);
      return;
    }
  }
};

var mobImg = new Image();
mobImg.src = "assets/Old enemies.png";
var MobSprite = class MobSprite extends PlayerSprite {
  static draw(mob) {
    if(mob.d === undefined) {
      mob.d = {
        img: mobImg,
        afn: 0,
        frn: 0
      };
    }
    mob.d.frn += Math.floor(100/delta);
    if(mob.d.frn > 40) {
      mob.d.afn++;
      mob.d.frn -= 40;
    }
    var mobDir = mob.aiData.dir,dirx;
    if(mobDir === undefined) {
      mobDir = -1;
    }
    var dirX = 56*(mobDir+1);
    var imgY = 0;
    if(currentLevel == 3) {
      imgY = 16;
    }
    if(mob.ai.name == "VerticalPatrolAI") {
      imgY = 48;
    }
```

```
    if(mob.ai.name == "Boss1AI") {
      imgY = 32;
    }
    switch(mob.d.afn % 5) {
      case(0):
        c.drawImage(mob.d.img,dirX,imgY,16,16,mob.x,mob.y,mob.w,mob.h);
        break;
      case(1):

c.drawImage(mob.d.img,dirX+16,imgY,16,16,mob.x,mob.y,mob.w,mob.h);
        break;
      case(2):

c.drawImage(mob.d.img,dirX+32,imgY,16,16,mob.x,mob.y,mob.w,mob.h);
        break;
      case(3):

c.drawImage(mob.d.img,dirX+48,imgY,16,16,mob.x,mob.y,mob.w,mob.h);
        break;
      case(4):

c.drawImage(mob.d.img,dirX+64,imgY,16,16,mob.x,mob.y,mob.w,mob.h);
        break;
      }
  }
};

var dirt = new RepeatingTile("assets/dirt2.png");

var level1Bg = new Background([
  new Sprite("assets/sky2-100by20.png",0,0,100,20,0,0,10000,2000,0.2),
  new Sprite("assets/sky.png",0,0,40,20,0,0,4000,2000,0)
]);
var level2Bg = new Background([
  new Sprite("assets/sky2-2.png",0,0,100,20,0,0,10000,2000,0.2),
  new Sprite("assets/sky1-2.png",0,0,40,20,0,0,4000,2000,0)
]);

var level3Bg = new Background([
  new Sprite("assets/sky2-100by20.png",0,0,100,20,0,0,10000,2000,0.2),
  new Sprite("assets/dark sky-2.png",0,0,40,20,0,0,4000,2000,0)
]);
var level4Bg = new Background([
  new Sprite("assets/nightSky.png",0,0,40,20,0,0,4000,2000,0)
]);
```

```
var level1Fg = new Background([
  new Sprite("assets/levels-1.png",0,0,800,80,0,0,20000,2000,1)
]);

var level2Fg = new Background([
  new Sprite("assets/levels-2.png",0,0,800,80,0,0,20000,2000,1)
]);

var level3Fg = new Background([
  new Sprite("assets/levels-3.png",0,0,800,80,0,0,20000,2000,1)
]);


var level4Fg = new Background([
  new Sprite("assets/levels-4.png",0,0,800,80,0,0,20000,2000,1)
]);

var AlexSprite = new PlayerSprite("assets/spritesheetFinal.png",{
  sw: 25,
  sh: 50,
  drx: 0,
  dry: 50,
  dlx: 25,
  dly: 50,
  urx: 50,
  ury: 0,
  ulx: 75,
  uly: 0,
  lbx: 75,
  lby: 0,
  rbx: 50,
  rby: 0,
  l1x: 75,
  l1y: 0,
  l2x: 25,
  l2y: 50,
  r1x: 50,
  r1y: 0,
  r2x: 0,
  r2y: 50,
  slx: 25,
  sly: 0,
  srx: 0,
  sry: 0,
```

```
  frn: 0,
  afn: 0
});

var MaxSprite = new PlayerSprite("assets/spritesheetFinal2.png",{
  sw: 25,
  sh: 50,
  drx: 25,
  dry: 50,
  dlx: 50,
  dly: 50,
  urx: 50,
  ury: 0,
  ulx: 0,
  uly: 50,
  lbx: 0,
  lby: 50,
  rbx: 50,
  rby: 0,
  l1x: 0,
  l1y: 50,
  l2x: 50,
  l2y: 50,
  r1x: 50,
  r1y: 0,
  r2x: 25,
  r2y: 50,
  slx: 25,
  sly: 0,
  srx: 0,
  sry: 0,
  frn: 0,
  afn: 0
});

var coin = new Tile("assets/crate.png",0,0,100,100);
```

switches.js

```
//Depends on module "platform"
if(modules.indexOf("platform") == -1) {
  throw "DependancyError: Module platform is required for Module
switch";
} else {
  //Push "switch" to list of included modules
```

```
  modules.push("switch");

  var Switch = class Switch extends Box {
    constructor(x,y,sprite1,sprite2,defaultPosition,entity) {
      super(x,y,1,2,false);
      this.on = defaultPosition;
      this.defaultPosition = defaultPosition;
      this.entity = entity;
      this.s1 = sprite1;
      this.s2 = sprite2;
    }

    draw() {
      if(this.open) {
        this.s1.draw(this.x,this.y,this.w,this.h);
      } else {
        this.s2.draw(this.x,this.y,this.w,this.h);
      }
    }

    collision(obj,dir) {
      if(obj.constructor.name  == "Player") {
        var triggered;
        if (keys[obj.controls.open]) {
          if (!triggered) {
            //Do something depending on the direction the collision
happened from.
            this.on = !this.on;
            this.entity.update();
            triggered = true;
          }
        } else {
            triggered = false;
        }
      }
    }

    reset() {
      this.on = this.defaultPosition;
    }
  };
  Switch.type = "dynamic";
}
```

Levels of Game:

World 1 Level 1 (w1l1.js)

```
var w1l1;

w1l1 = new Level(200,level1Bg,level1Fg);

w1l1.statics.push(
  new Platform(-3, 0, 3, 20, noTile),
  new Platform(0, 18, 53, 2, noTile),
  new Platform(35, 17, 18, 1, noTile),
  new Platform(36, 16, 17, 1, noTile),
  new Platform(37, 15, 8, 1, noTile),
  new Platform(38, 14, 7, 1, noTile),
  new Platform(39, 13, 6, 1, noTile),
  new Platform(58, 16, 13, 4, noTile),
  new Platform(60, 13, 11, 3, noTile),
  new Platform(74, 17, 2, 3, noTile),
  new Platform(79, 13, 4, 7, noTile),
  new Platform(91, 13, 4, 7, noTile),
  new Platform(95, 14, 1, 1, noTile),
  new Platform(95, 15, 2, 1, noTile),
  new Platform(95, 16, 3, 1, noTile),
  new Platform(95, 17, 4, 1, noTile),
  new Platform(95, 18, 5, 1, noTile),
  new Platform(95, 19, 27, 1, noTile),
  new Platform(116, 17, 2, 3, noTile),
  new Platform(120, 15, 2, 5, noTile),
  new Platform(124, 13, 2, 7, noTile),
  new Platform(126, 17, 2, 3, noTile),
  new Platform(128, 11, 2, 9, noTile),
  new Platform(130, 15, 2, 5, noTile),
  new Platform(132, 9, 2, 11, noTile),
  new Platform(140, 18, 2, 1, noTile),
  new Platform(148, 15, 12, 5, noTile),
  new Platform(162, 13, 2, 2, noTile),
  new Platform(166, 13, 2, 2, noTile),
  new Platform(170, 18, 30, 2, noTile),
  new Platform(170, 17, 11, 1, noTile),
  new Platform(170, 16, 10, 1, noTile),
  new Platform(170, 15, 9, 1, noTile),
  new Platform(170, 14, 8, 1, noTile),
  new Platform(170, 13, 7, 1, noTile),
  new Platform(170, 12, 6, 1, noTile),
  new Platform(170, 11, 5, 1, noTile),
```

```
  new Platform(170, 10, 4, 1, noTile),
  new Platform(192, 11,  2,  2, noTile),
  new Platform(195,  0,  5, 9, noTile),
  new Platform(195, 15,  5,  9, noTile),
  new Platform(196, 9,  4,  1, noTile),
  new Platform(196, 14,  4,  1, noTile)
);

console.log("World-1 Level-1 (w1l1.js) Loaded");
```

World 1 Level 2 (w1l2.js)

```
var w1l2;

w1l2 = new Level(200,level2Bg,level2Fg);

w1l2.add(
  new Platform(  0, 14,  2,  1, noTile),
  new Platform(  0, 15,  3,  1, noTile),
  new Platform(  0, 16,  4,  1, noTile),
  new Platform(  0, 17,  5,  1, noTile),
  new Platform(  0, 18, 46,  2, noTile),
  new Platform( 14, 14,  7,  1, noTile),
  new Platform( 30, 17, 16,  1, noTile),
  new Platform( 31, 16, 15,  1, noTile),
  new Platform( 32, 15, 14,  1, noTile),
  new Platform( 33, 14, 13,  1, noTile),
  new Platform( 34, 13, 12,  1, noTile),
  new Platform( 35, 12, 11,  1, noTile),
  new Platform( 36, 11, 10,  1, noTile),
  new Platform( 37, 10,  9,  1, noTile),
  new Platform( 38,  9,  8,  1, noTile),
  new Platform( 39,  8,  7,  1, noTile),
  new Platform( 40,  7,  6,  1, noTile),
  new Platform( 41,  6,  5,  1, noTile),
  new Platform( 42,  5,  4,  1, noTile),
  new Platform( 54, 13, 19,  7, noTile),
  new Platform( 81, 13,  1,  7, noTile),
  new Platform( 95, 18, 20,  2, noTile),
  new Platform(107, 17,  8,  1, noTile),
  new Platform(108, 16,  7,  1, noTile),
  new Platform(109, 15,  6,  1, noTile),
  new Platform(110, 14,  5,  1, noTile),
  new Platform(111, 13,  4,  1, noTile),
```

```
  new Platform(114, 19, 47,  1, noTile),
  new Platform(138, 13,  2,  1, noTile),
  new Platform(138, 14,  3,  1, noTile),
  new Platform(138, 15,  4,  1, noTile),
  new Platform(138, 16,  5,  1, noTile),
  new Platform(138, 17,  6,  1, noTile),
  new Platform(138, 18, 23,  1, noTile),
  new Platform(170, 17, 11,  1, noTile),
  new Platform(170, 18, 30,  2, noTile),
  new Platform(170, 16, 10,  1, noTile),
  new Platform(170, 15,  9,  1, noTile),
  new Platform(170, 14,  8,  1, noTile),
  new Platform(170, 13,  7,  1, noTile),
  new Platform(170, 12,  6,  1, noTile),
  new Platform(170, 11,  5,  1, noTile),
  new Platform(170, 10,  4,  1, noTile),
  new Platform(192, 11,  2,  2, noTile),
  new Platform(195,  0,  5, 9, noTile),
  new Platform(195, 15,  5,  9, noTile),
  new Platform(196, 9,  4,  1, noTile),
  new Platform(196, 14,  4,  1, noTile),
  new BouncingPlatform(46,17,8,3, noTile),
  new BouncingPlatform(76,16,2,1, noTile),
  new BouncingPlatform(87,19,3,1, noTile),
  new MovingPlatform(123,13,6,2, dirt, PatrolAI, {x1:115, x2:138, vX:1,
dir:1}),
  new MovingPlatform(150,14,3,1, dirt, PatrolAI,{x1:146, x2:157, vX:1,
dir:1}),
  new MovingPlatform(160,11,3,1, dirt, PatrolAI,{x1:156, x2:167, vX:3,
dir:-1}),
  new MovingPlatform(170,8,3,1, dirt, PatrolAI,{x1:166, x2:177, vX:2,
dir:1})
);
console.log("World-1 Level-2 (w1l2.js) Loaded");
```

World 1 Level 3 (w1l3.js)

```
var w1l3;

w1l3 = new Level(200,level3Bg,level3Fg);

w1l3.add(
  new Platform(  0, 14,  7,  6, noTile),
  new Platform( 10, 17, 40,  3, noTile),
```

```
  new Platform( 40, 13, 10,  4, noTile),
  new Platform( 60, 13,  7,  7, noTile),
  new Platform( 74, 13,  7,  7, noTile),
  new Platform(104,  6,  8, 14, noTile),
  new Platform(126,  6,  4, 14, noTile),
  new Platform(154,  9,  6, 11, noTile),
  new Platform(170, 17, 11,  1, noTile),
  new Platform(170, 18, 30,  2, noTile),
  new Platform(170, 16, 10,  1, noTile),
  new Platform(170, 15,  9,  1, noTile),
  new Platform(170, 14,  8,  1, noTile),
  new Platform(170, 13,  7,  1, noTile),
  new Platform(170, 12,  6,  1, noTile),
  new Platform(170, 11,  5,  1, noTile),
  new Platform(170, 10,  4,  1, noTile),
  new Platform(192, 11,  2,  2, noTile),
  new Platform(195,  0,  5,  9, noTile),
  new Platform(195, 15,  5,  9, noTile),
  new Platform(196,  9,  4,  1, noTile),
  new Platform(196, 14,  4,  1, noTile),
  new Mob(24,15,2,2,PatrolAI,{dmg:10,x1:10,x2:40,vX:1.6,dir:1,}),
  new Mob(53.5,7,3,3,VerticalPatrolAI,{dmg:15,y1:5,y2:19,vY:1.6,dir:1}),
  new Mob(69,7,3,3,VerticalPatrolAI,{dmg:15,y1:5,y2:19,vY:1.6,dir:-1}),
  new Mob(86,13,1,1,NoAI,{dmg:5,}),
  new Mob(99,9,1,1,NoAI,{dmg:5,}),
  new Mob(117,10,2,2,VerticalPatrolAI,{dmg:10,y1:1,y2:16,vY:4,dir:1}),
  new Mob(163,16,4,4,VerticalPatrolAI,{dmg:20,y1:9,y2:19,vY:2,dir:-1}),

  new MovingPlatform(85,14,3,1, dirt,
PatrolAI,{x1:83,x2:101,vX:5,dir:1}),
  new MovingPlatform(98,10,3,1, dirt,
PatrolAI,{x1:92,x2:101,vX:5,dir:-1}),
  new MovingPlatform(132,10,3,1, dirt,
VerticalPatrolAI,{y1:6,y2:19,vY:1,dir:1}),
  new MovingPlatform(140,10,3,1, dirt,
VerticalPatrolAI,{y1:6,y2:19,vY:10,dir:-1}),
  new MovingPlatform(148,10,3,1, dirt,
VerticalPatrolAI,{y1:9,y2:19,vY:7,dir:1})
);
console.log("World-1 Level-3 (w1l3.js) Loaded");
```

World 1 Boss (w1boss.js)

```
var w1boss;
```

```
w1boss = new Level(80,level4Bg,level4Fg);

w1boss.add(
  new Platform(0,  18, 80, 2,  noTile),
  new Platform(30, 14, 10.01, 6,  noTile),
  new Platform(40, 14, 1, 1,  noTile),
  new Platform(40, 0,  1,  11,  noTile),
  new Platform(72, 14, 8,  1,  noTile),
  new Platform(80, 0, 1, 14,  noTile),

  new Door(40,11,1,3,dirt,true,0),
  new Door(74,15,1,3,dirt,false,1)
);

var bossPlat = new MovingPlatform(72,13,8,1, dirt,
PatrolAI,{x1:41.05,x2:79.95,vX:0,dir:-1});

var bossMob =  new
Mob(72,6,8,8,Boss1AI,{dmg:50,health:4,locked:false},Boss1Hit);

w1boss.dynamics.push(
 bossMob,
 bossPlat
);

w1boss.reset = function() {
  bossMob.aiData = {dmg:50,health:4,locked:true,stage:undefined};
  bossMob.x = 32*u;
  bossMob.y = 6*u;
  bossPlat.x = 32*u;
  bossPlat.aiData.vX = 0;
};

console.log("World 1 Boss Level (w1boss.js) Loaded");
```