Mobile Games Development Blog

Contents

[Before you read 1](#_Toc61087262)

[08/01/2021 1](#_Toc61087263)

[Game Introduction 1](#_Toc61087264)

[Game Mechanics 2](#_Toc61087265)

[Classes 2](#_Toc61087266)

[Class aSprite 3](#_Toc61087267)

[Class background 4](#_Toc61087268)

[Creating an Environment 4](#_Toc61087269)

[Player Class 7](#_Toc61087270)

[The Heat Ray 8](#_Toc61087271)

[Game states 9](#_Toc61087272)

[References 10](#_Toc61087273)

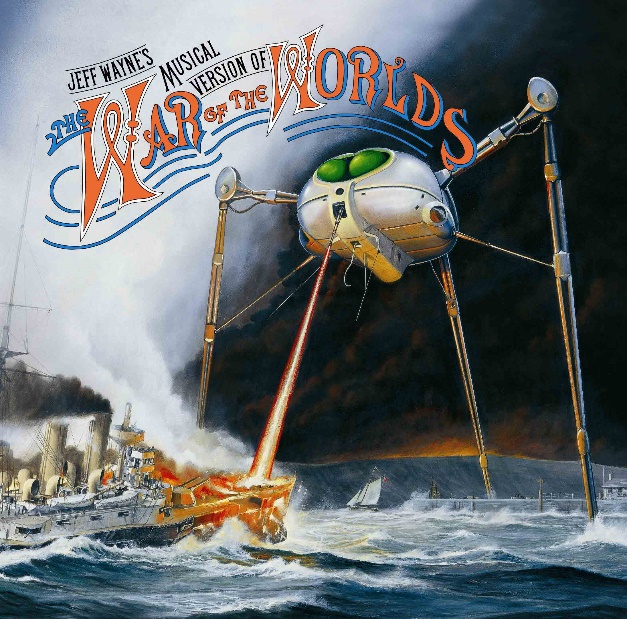
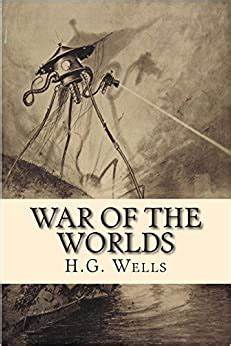
# Before you read

This blog makes use of animated gifs to help the reader understand points. These gifs may not animate on mobile devices.

# 08/01/2021

## Game Introduction

For my mobile games project, I wish to create an homage to my favourite musical (WAYNE, 1978) and novel (WELLS, 1898) War of the Worlds. In the game you will control the heat ray of a Martian fighting machine as it walks through the English countryside targeting humans, artillery and eventually the HMS Thunderchild in an attempt to take over the world (England).

**Figure 1 Cover for The musical version of WOTW Figure 2 Cover for the novel version of WOTW**

## Game Mechanics

The player will not move instead the world around them leaving them to control the heat ray by tapping in front of the fighting machine. Once tapped a red line will be rendered from the nozzle of the heat ray to the point the player has tapped on, if the Martian has shot an enemy the appropriate actions will be taken, and score will be added. The player will be running against the clock because just like in the story the Martians are vulnerable to bacteria and the player will need to reach a certain score before they are claimed by disease. Should the player reach the score limit they will fight the HMS Thunderchild and should they win they will take the planet.

## Classes

Looking at figure 3 we can see a breakdown of the classes in the game. All instances will inherit from the class aSprite as they must poses an x value, a y value, velocity, a sprite and a scale. The humans artillery and Thunderchild will all inherit from the enemy class as they al must be able to die and set a tag and type.

Timeline

Description automatically generated

**Figure 3 Class Diagram**

## Class aSprite

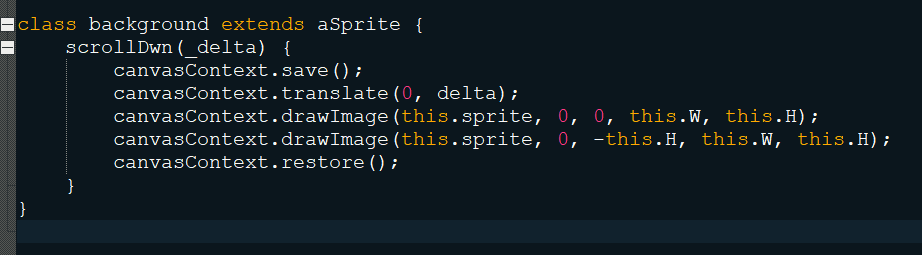
The class aSprite is the base for all objects within the game and contains information including the objects position, scale, velocity, render code and its collision status with other objects. Its code can be seen in figure 4.



**Figure 4 class aSprite**

## Class background

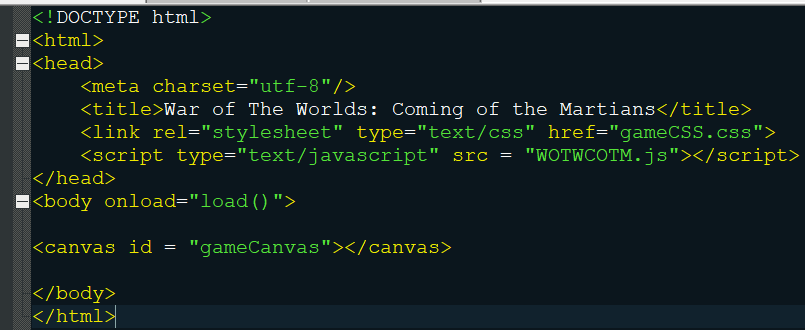
The background class extends from aSprite so it includes all the functions and variables that belong to it. The only function that is exclusive to background is the scrollDwn function which allows the background to give the impression it is moving. The code is taken from one of the labs completed in this course and modified from the image scrolling horizontally to vertically this code can be seen in figure 5. With the current code created we can create our environment.



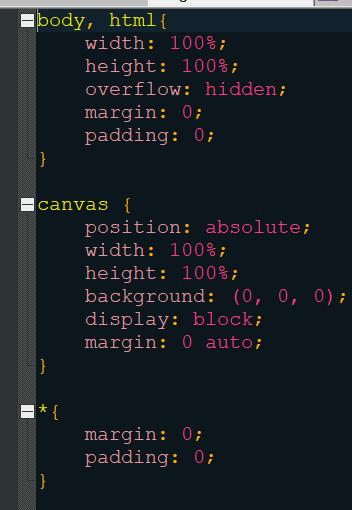
**Figure 5 scroll function.**

## Creating an Environment

This project combines a JavaScript file along with both a HTML and CSS file in order to run (the HTML file and CSS can be seen in figure 6 and 7). Setting up the game world requires several steps; the first

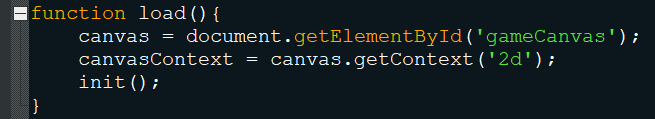


**Figure 6 game HTML**



**Figure 7 game CSS**

function ‘load’ (figure 8) gathers the canvas and canvas context from the HTML file followed by calling the init function.



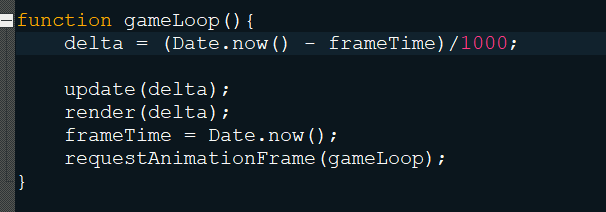
**Figure 8 Load Function**

The ‘init’ function (figure 9) adds event listeners to the window and canvas so that the game can react to user input. The ‘addEventListener’ function works by quoting an event type for the program to check for followed by a function for the program to do once that even happens. It is also in our ‘init’ function that we create our static game objects in this case the bckgrnd object. Finally the init function starts the frame time variable ajd calls the gameLoop for the first time.



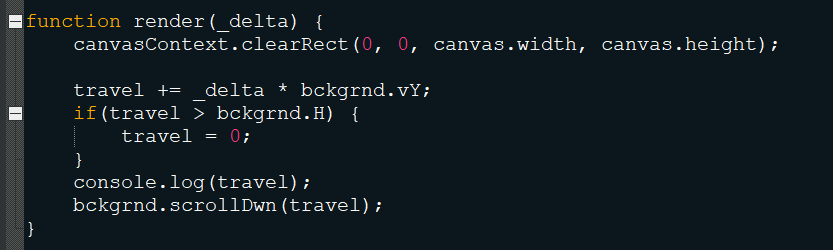
**Figure 9 init Function**

The ‘gameLoop’ is a recursive function that calls itself at the end of its calculations (figure 10). Inside the loop it calls the update and render function and finishes by updating the frame time.



**Figure 10 gameLoop Function**

The render function is responsible for drawing all the entities on the screen (figure 11).



**Figure 11 render Function**

The above code creates an effect of a background moving downwards (Figure 12) which when combined with the fighting machine sprite will make it look like is moving.

**A picture containing text, electronics

Description automatically generated**

**Figure 12 the moving background**

## Player Class

The player class holds the players health along with the means to render the health bar. In the init function (Figure 9) a new player is created called ‘fightingMachine’ along with 0-2 of the array ‘playerHealth’ which are then rendered in the render function (Figure 11). It creates the effects seen in figure 13.

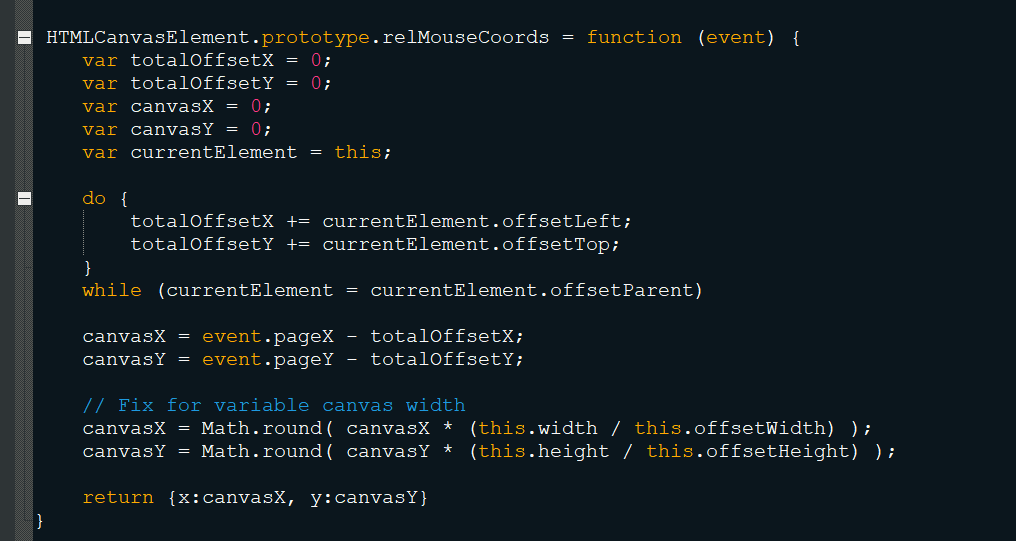
A screen shot of a video game

Description automatically generated with medium confidence

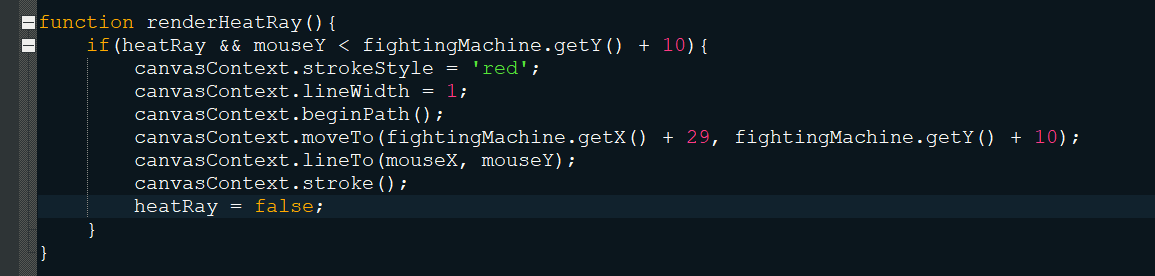
**Figure 13 Player Render**

## The Heat Ray

The heat ray is the method the player uses to attack its enemies. In order to shoot the heat ray the player will tap on the screen to where they want to shoot the heat ray. The heat ray will be rendered using the line render tools available in JavaScript. In order to calculate the mouse I’m using code given to us in a lab which finds the mouse positions in relation to the canvas (figure 14). The heat ray function sets the canvas context variables to what is needed for the heat ray to render as a red line the code can be seen in figure 15. The resulting code can be seen in figure 16.



**Figure 14 Code given in lab 6**



**Figure 15 Render Heat Ray code**

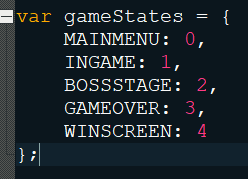
**A picture containing text, electronics

Description automatically generated**

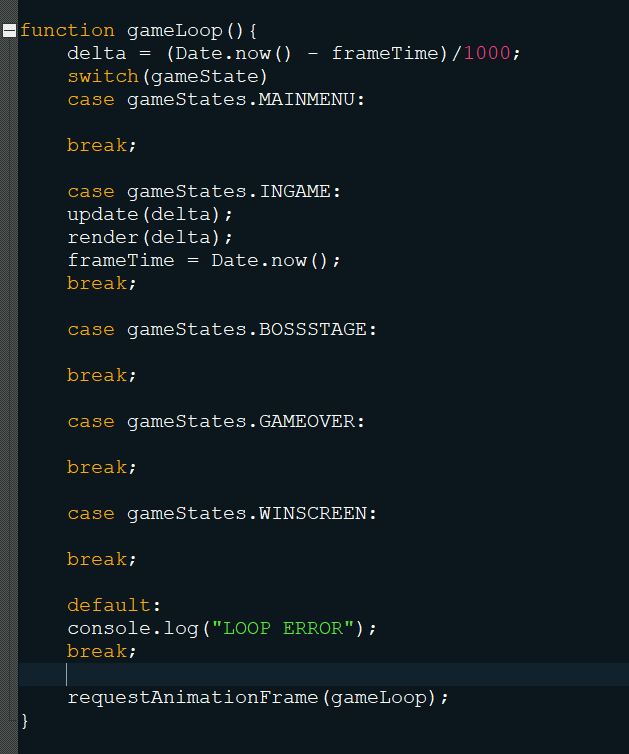
**Figure 16 Fighting Machine firing the heat ray**

## Game states

In order for the game to know what stage of the game it is at i.e., main menu, in game, boss stage etc an enum variable has been created so that the game loop can be broken up (Figure 17). In order to do the correct code at the appropriate time a switch statement is added into the game loop (Figure 10) this can be seen in Figure 18.



**Figure 17 Game state enum**



**Figure 18 New Game Loop**

# 09/01/2021

## Button Class

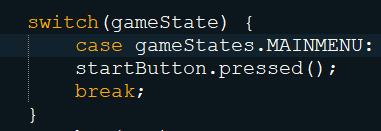
The button class is responsible for storing the tag variable and the pressed functions for the buttons. The code for it can be seen in figure 19.



**Figure 19 button class**

## Creating a start button

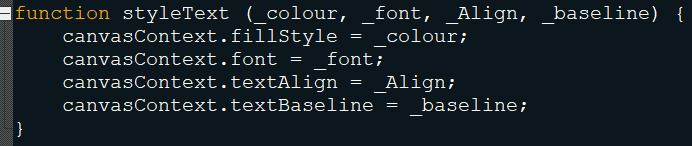
In order to transition from the main menu state to the in game state (Figure 18) a button is needed. So in the init function (Figure 9) a new button object is created and the tag is set to start. Then once the screen is touched the program checks the game state and calls the pressed function for the appropriate button (Figure 20).



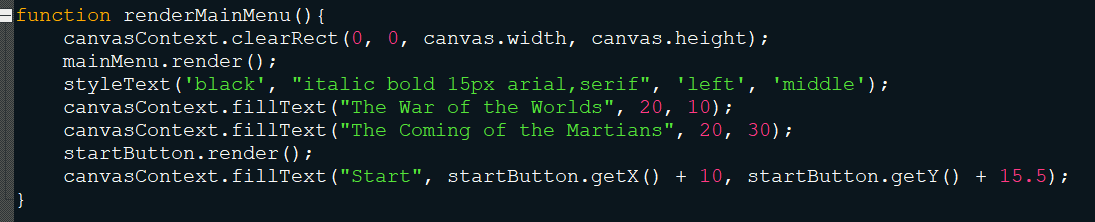
**Figure 20 button pressing**

## Creating a main menu

So that the player has something to look at before starting the game a main menu is needed. First off a background is needed for the menu so a new background object is created. Secondly a render function is made to be called in the main menu state of the game loop (Figure 18). The code inside the render function clears the screen, renders the menu, renders text using a new function called style text (Figure 21) which sets the colour, font, alignment and baseline of the text. Finally, the render function renders the start button and the text for it (Figure 22). The resulting code creates the resulting main menu in figure 23.



**Figure 21 Style Text**



**Figure 22 render main menu**

**Qr code

Description automatically generated with medium confidence**

**Figure 23 Main menu**

## Enemy Class

The enemy class acts as a parent class for all enemies in the game and holds the code for everything that an enemy can do; set a tag, set a type and die. In order to calculate the death of enemies the game runs a switch statement on the type variable of enemy and then performs the appropriate calculations, this can be seen in Figure 24.



**Figure 24 Enemy Class**

## Humans

The first enemy to be created is a simple human. First the game needs to know when to spawn one. To do this the game needs two variables a start time and a counter variable in the update function the counter gets increased by the value of the current date minus the starter date divided by one thousand. Once the counter passes 0.75 it resets both the counter to zero and the start time to the current date. The human class inherits from the enemy class, being part of the human class allows for its members to set the y velocity. This is needed to give a scattering effect for the humans. To spawn a human wo calculations, have to be made fist of all the game has to pick what side of the screen to spawn them on, this is done by a random number generator between one and two. Secondly a skin must be decided upon. The file architecture of the project allows for an image to be loaded using a number, so a random number between one and the highest number of skins in generated. This data then combines to allow the creation of a human. Upon creation of the human a tag and type must be assigned so it can be recognised along with the calculation of a y velocity. In order for the human to move the game then loops through the humans array an updates their x and y values. This code and resulting effect can be seen in figure 25 and 26.



**Figure 25 human Spawn code**

**A picture containing text, computer, computer

Description automatically generated**

# References

WAYNE, J., 1978.*Jeff Wayne's Musical Version of The War of the Worlds .*New York City: CBS Records, 9.06.

WELLS, H.G., 1898.*War of the Worlds.*William Heinemann of London.