Spencer Dallas
ARCX Inc
July 2018
Revision 6

Space Invaders Tutorial

Contents

Demo the Game ... 3
Development Environment ... 4
Server Hosting ... 7
- Internet Information Services ... 7
- FlexFrame ... 11
- Wiring the Joysticks ... 14
- Powering the Box ... 18
  - Standard Power ... 18
  - Power Over Ethernet ... 20
  - Connecting the box to IIS ... 22

Code ... 24
- 001: HTML and CSS ... 25
- 002: Variable Declaration ... 26
- 003: Movement Variables ... 28
- 004: Sprite Arrays ... 30
- 005: Objects ... 34
- 006: Start Game ... 36
- 007: Starting and Stopping Timers ... 38
- 008: Background Display ... 40
- 009: Player Movement ... 41
- 010: Finding Pressed Buttons ... 42
- 011: Shoot Function ... 43
- 012: Alien Movement ... 44
- 013: Hit Registration ... 46
- 014: Score Handling ... 48
- 015: Barrier Decay ... 49
- 016: Alien Shooting ... 50
- 017: Drawing Aliens and Barriers ... 51
- 018: Bullet Movement ... 52
- 019: Endgame Conditions ... 53
- 020: Winning and Losing ... 54
- 021: Restart Conditions ... 56
- 022: Sprite Animation ... 57
- 023: Collision Function ... 59
- 024: Sprite Drawing ... 61

Introduction

The purpose of this project is to get a better understanding of developing applications for the ARCX Inc Logic Controllers. This document discusses developing Web Automation Toolkit (WAT) applications on your local machine, staging and debugging the application locally and finally deploying the application to a target ARCX Device. WAT is an API and series of development languages using HTML5, JavaScript and CSS3.

Demo the Game

Start by running a sample of the game in our local browser. It is recommended to use Google Chrome Browser.

Enter the following URL (this code is adapted to work on a computer, not on the UP3K)

https://support.arcx.com/demos/SpaceInvaders/SpaceInvaders.html

In playing the game one gets a feel for all the moving parts. Notice the following;

- Movement and pattern of the aliens and the player
- the breaking of the barriers increases the score
- the death of each alien does not affect the next, even though they all move together
- the bullets starting at the player or alien become their own object and flying away from the player or alien
- how the player ship breaks when it dies, the aliens explode, and the barriers erode
- smaller details, such as how a bullet will pass through an exploding alien but still stop at an eroding barrier.

Once an understanding of the game is gained, the source code will be much easier to understand.

Development Environment

Next, set up the components needed to code the application.

Install a text editor. Notepad++ is recommended:
https://notepad-plus-plus.org/download/v7.5.7.html

Windows Notepad can be used, but without context highlighting, it is more difficult to find syntax errors. Other web development source code editors can be used and left as an exercise to the reader.

Once the editor is open, HTML is used to set up the framework for a webpage. The following is how you make a page with the text “HELLO WORLD” displayed.

Note: <html> is called an html tag. </html> is called a closing tag. This works for any contents of the tag. <p></p>, <body></body>, <script></script>, etc.

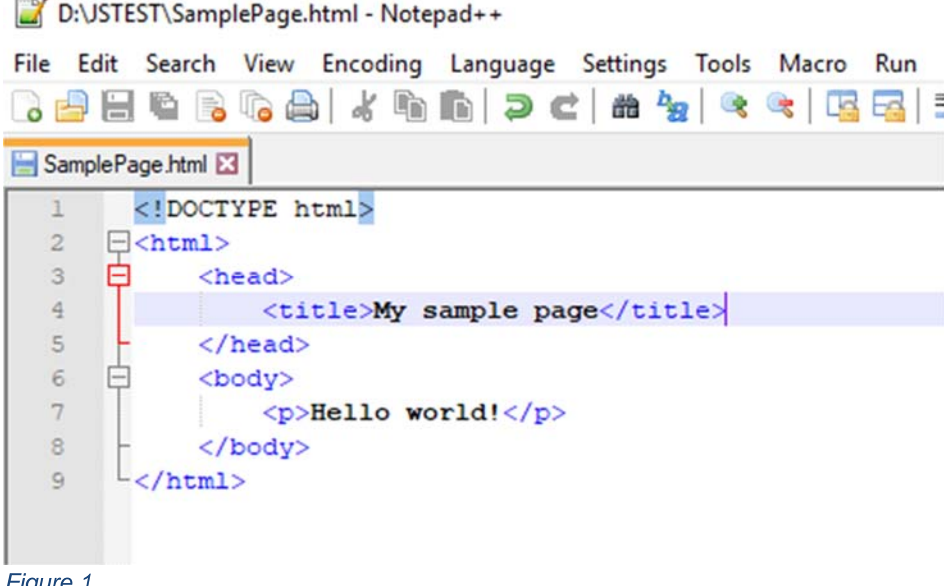


Figure 1

Save the text document as "SamplePage.html". This will tell the computer that it is an html file, and not a standard text file. After saving this, the tags will be highlighted. Now, open the saved file, and it will open a web page with the desired text.

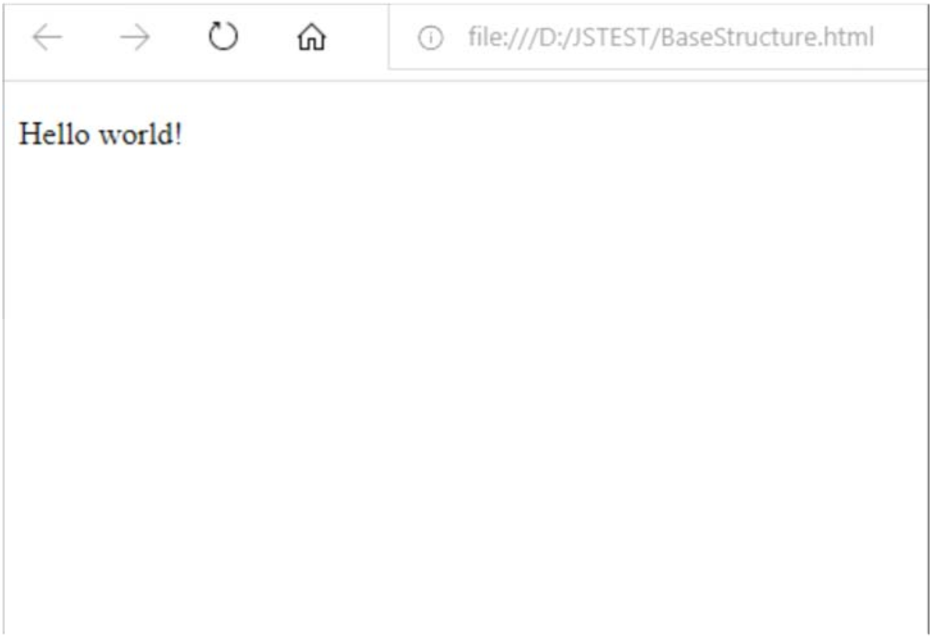


Figure 2

There is more function to a webpage than plain text, so a more complete framework would look like this:

Note: “//” means the code is commented out. This is used to describe code, but the commented does not actually run.

```
<html>                                      //starts the html page
        <head>                              //header, this does not show up but affects the body
                <title>
                        My Sample Page
                </title>
                <style>                     //starts CSS, affecting how the page is displayed

                </style>                    //closes the CSS
        </head>                             //closes the header
        <body>                              //this is where html code goes
                <p>
                        HELLO WORLD
                </p>
        </body>                             //ends where html code goes
        <script>                            //starts where javascript goes

        </script>                           //ends where javascript goes
</html>                                     //ends the html page
```

This page would output the same thing; it just includes the ability for CSS and JavaScript to be added to the code.

Almost all the code written for this game is in JavaScript (between the script tags), but there are some details written outside. HTML is like a framework; It sets up the structure for the rest of the code, and provides the basic resources to work with. CSS goes farther, allowing styling and more dynamic visuals to the page. It allows the programmer to directly interact with what appears on the page and change its appearance. JavaScript allows further function of the page. With it, the user is able to access more data in more interactive ways.

Write the complete framework notepad++ and save the text document as "SpaceInvaders.html".

Server Hosting

Why you need to host the application:

This project hosts a server using both Internet Information Services and FLEXframe. Hosting servers allows multiple users to access the program from different machines, as well as allows easier testing for code. They also allow deployment where users can run the code on several different types of machines (computer vs. UPPK) from the same place.

Internet Information Services

The first way to execute the code is through Internet Information Services. This runs through windows and is quick and simple to set up locally with no prior knowledge.

To set up IIS, go to the control panel. Click on "programs", click on "programs and features", and from there find "Turn Windows features on or off." From here, turn on Web Management Tools and World Wide Web Services. The computer may need to reboot to complete this process.

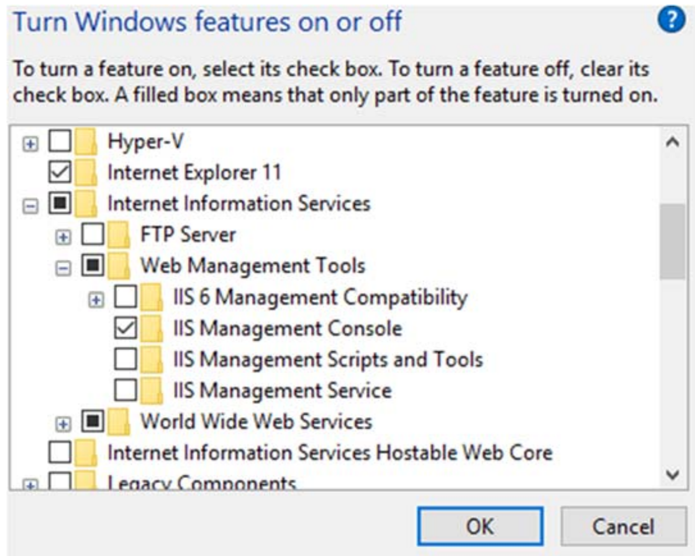


Figure 3

Now, if everything is set up correctly, a search for localhost in the browser, you should display this:

Show what you type in chrome here

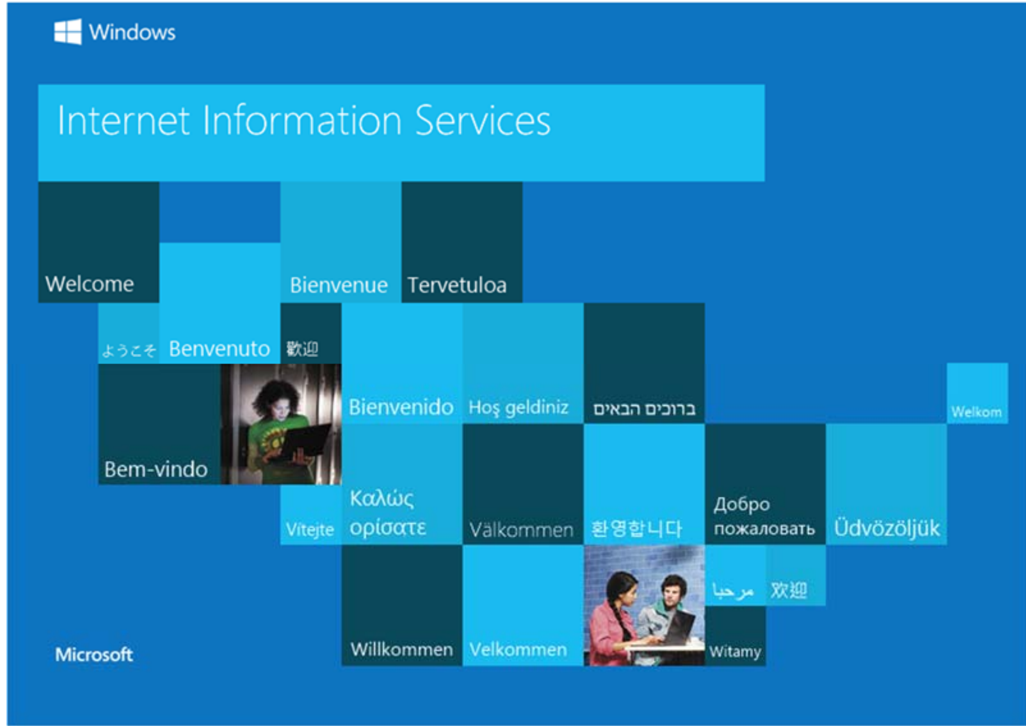


Figure 4

Now that this is active, go to IIS manager. This can be done by searching for internet information services in the windows search bar and checking for the manager. It should look like this:

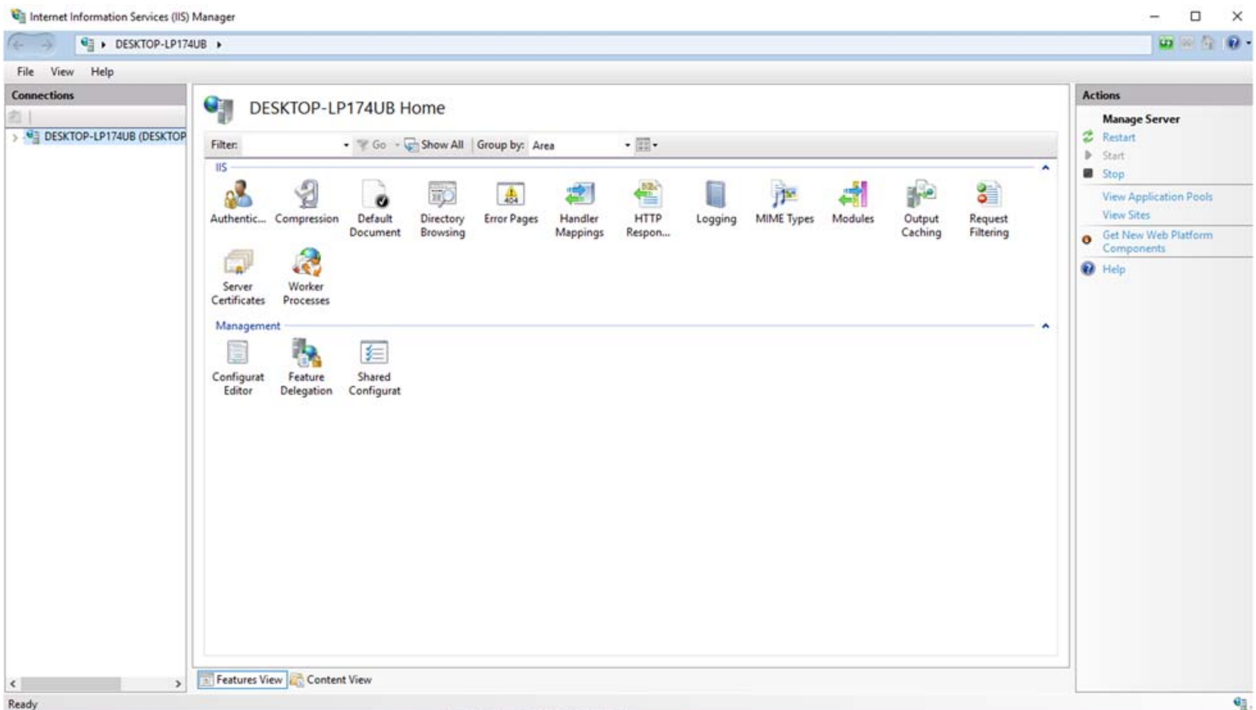


Figure 5

On the right side, under manage server, start the server, and make sure to restart the server whenever files are modified.

To run code on a server you will need to know/have:

- The computer's IP Address
- A file installed (hosted) on the server

The computer's IP will be first. To get the computer's IP, search ***cmd*** in the windows search bar. It will open a log. Type ipconfig, press enter, and look for IPv4. This is the local IP that will be used.

```
C:\Users\        >ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : arcx.com
   Link-local IPv6 Address . . . . . : fe80::9c62:78f:443f:fca8%4
   IPv4 Address. . . . . . . . . . . : 192.168.2.177
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.2.10
```

Figure 6

- In this example, the IP is 192.168.2.177. This may not match yours.

Now to run the chosen file, add it to the root folder, the default folder to be accessed when IIS is accessed by the computer. To do this, look for the root folder within C:\inetpub\wwwroot

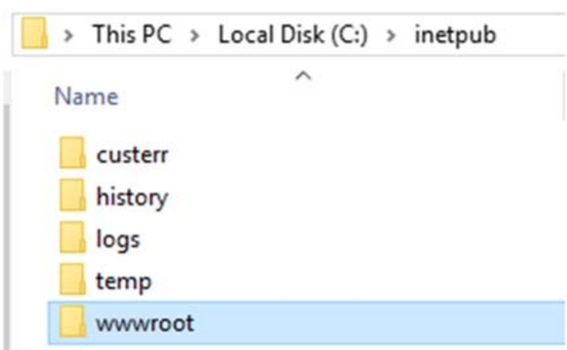


Figure 7

Place SpaceInvaders.html in this ***wwwroot*** folder.

Once complete, the file should look like this:

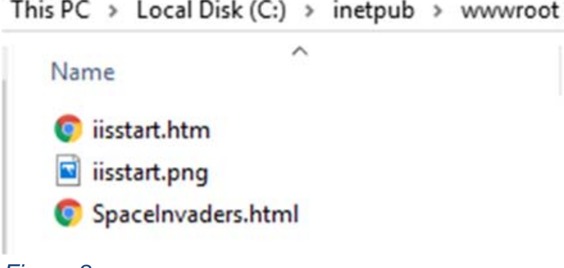


Figure 8

To run the code on the computer, search the computer's ip followed by the file name like this in the browser:

192.168.2.177/SpaceInvaders.html

Figure 9

Note: Use of the word local refers to the development machine as opposed to the target hardware device. If the file is being manipulated on the same machine as the server is being hosted, the IP can be replaced with "localhost". This would look like "localhost/SpaceInvaders.html"

FlexFrame

This code uses FlexFrame version 1.2.1, this code may not be valid in all future updates.

Open the flexframe installer on your desktop and install the program. To run the program, search for FlexFrame Command Prompt and open it. It will take you to this screen.

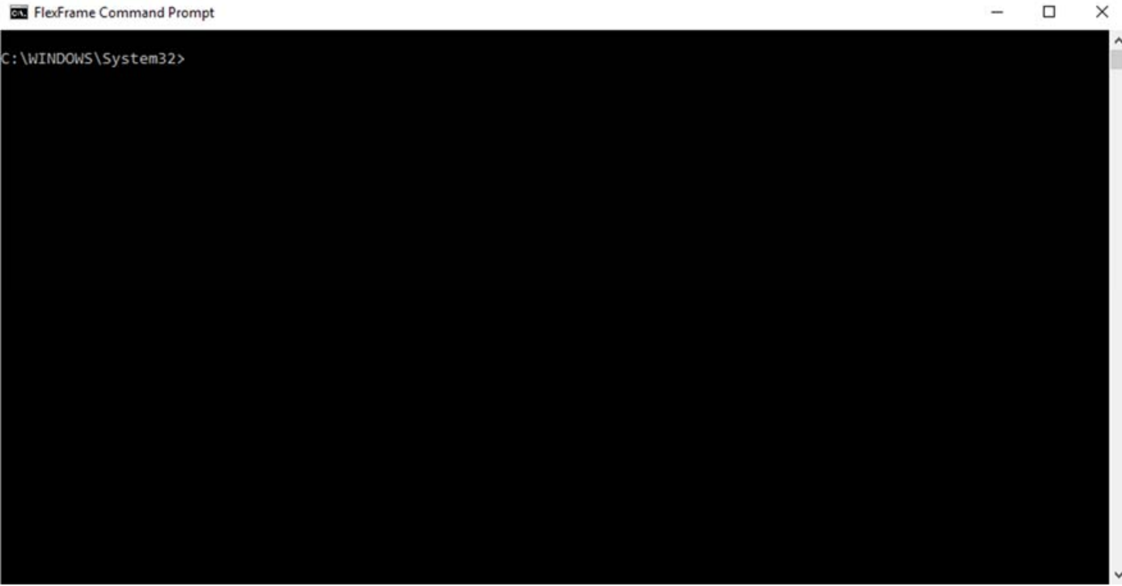


Figure 10

Now the directory must be pointed to the file you want to store your FlexFrame files. The command cd is used, followed by a directory. This example uses a test file on the desktop.

Figure 11

Now the FlexFrame console is pointing to the desired folder. Next to create all the initialization files necessary to run FlexFrame, use the command “flexframe config” and press enter when prompted. This will create files called “FlexFrame.json” and “index.pug.”

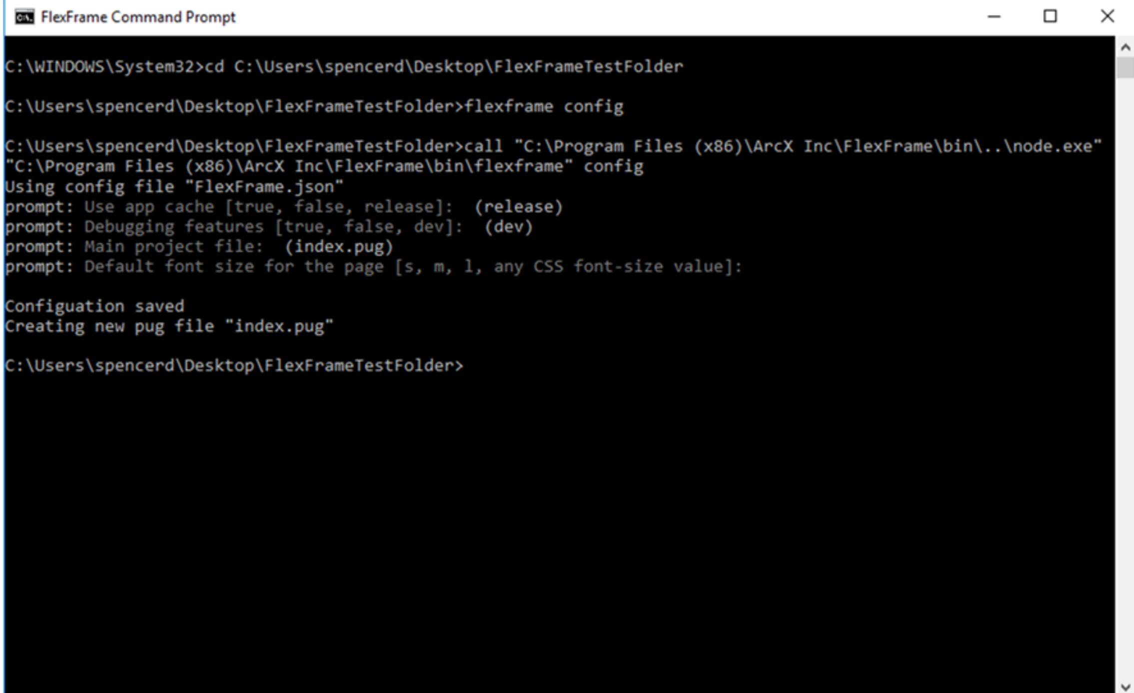
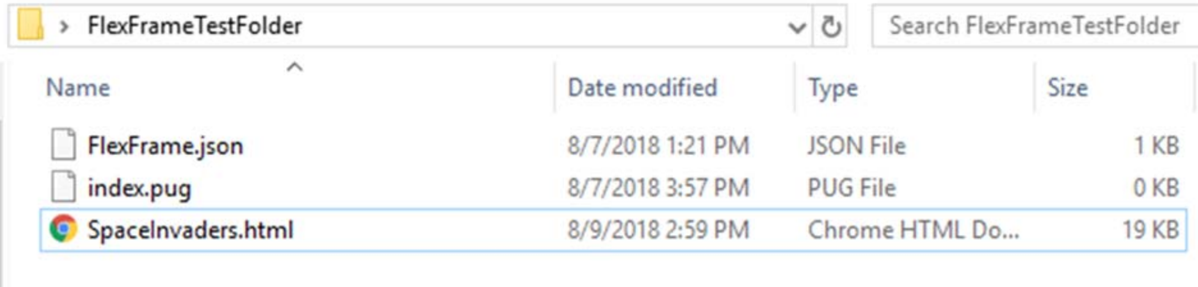
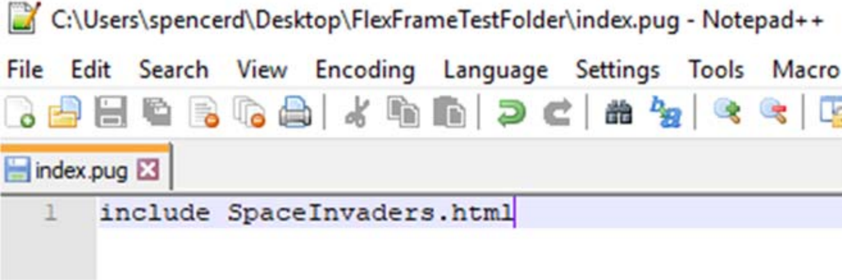


Figure 12

After this, place the ***SpaceInvaders.html*** file in the folder being accessed, so it looks like this:



Next, in the ***index.pug*** file, use notepad++ add a line to include *SpaceInvaders.html* in its running operation.



Finally, in the flexframe command prompt, use the command flexframe to host a server that runs the code in the index.pug file.

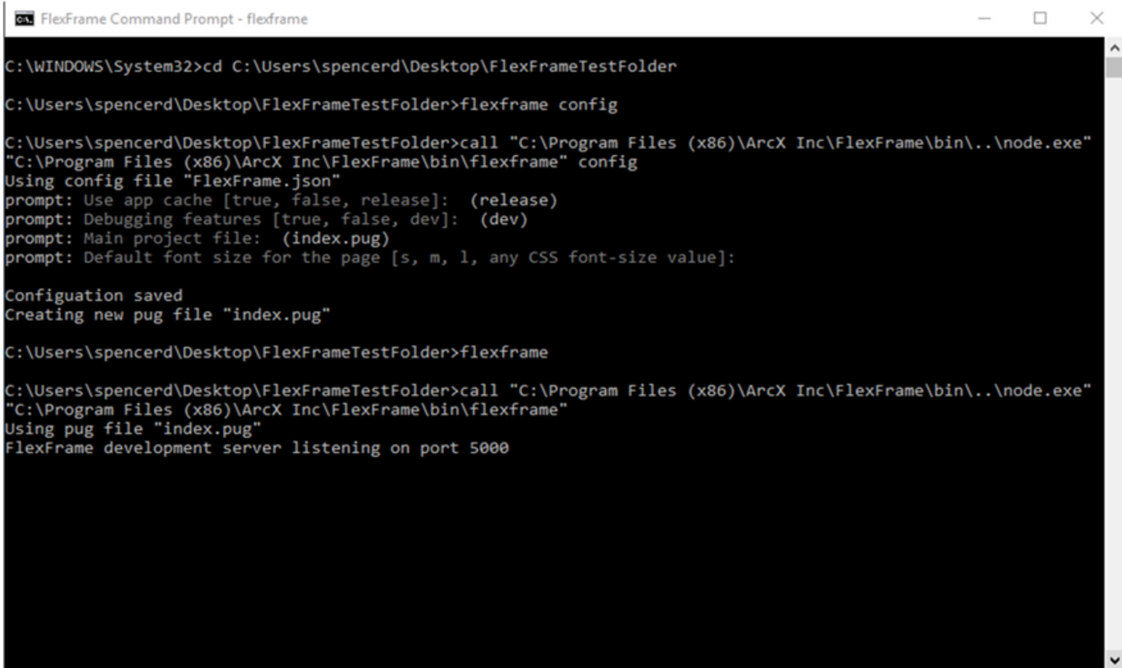


Figure 13

To access this server, similar to in IIS, the IP of the host machine is needed. Earlier it was established the example IP is 192.168.2.177, The other information needed is the port the server is hosted on, shown in the last line of the current FLEXframe command prompt. The default port is 5000, and that is what this example shows. To run the code, serch for the host IP followed by the port, like so:

192.168.2.177:5000

Wiring the Joysticks

The wiring can be done in many configurations of inputs, but this is the wiring setup that works with the provided code.

Figure 14

The joysticks connect to the inputs on the side below the screen. The wires from the joysticks plug into a terminal block which then plugs into the UP3K itself. This is the end product.

Figure 15

This is the orientation the box should be in for wiring. The joysticks should be upside down with the wires on the top left side towards the back.

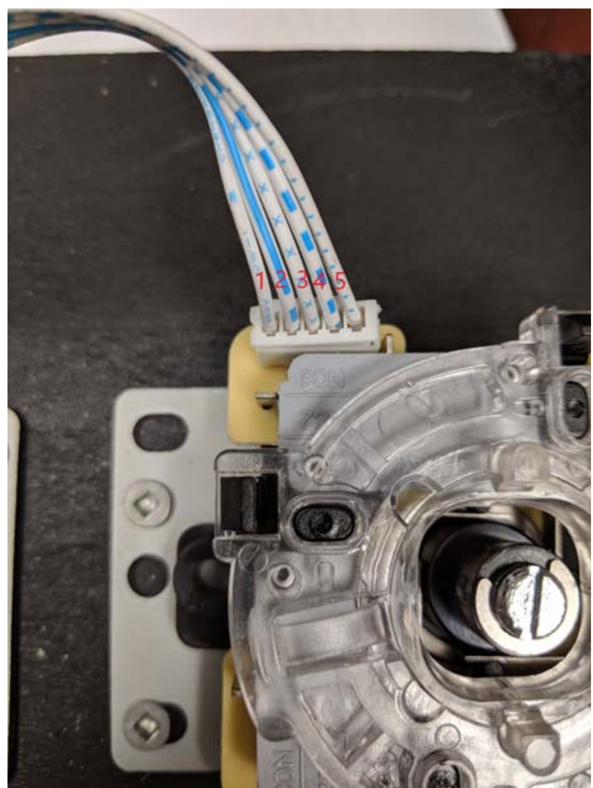


Figure 16

These are the wires labelled 1 through 5, 1 being the common (or ground) wire, and the rest representing a direction the joystick is held.

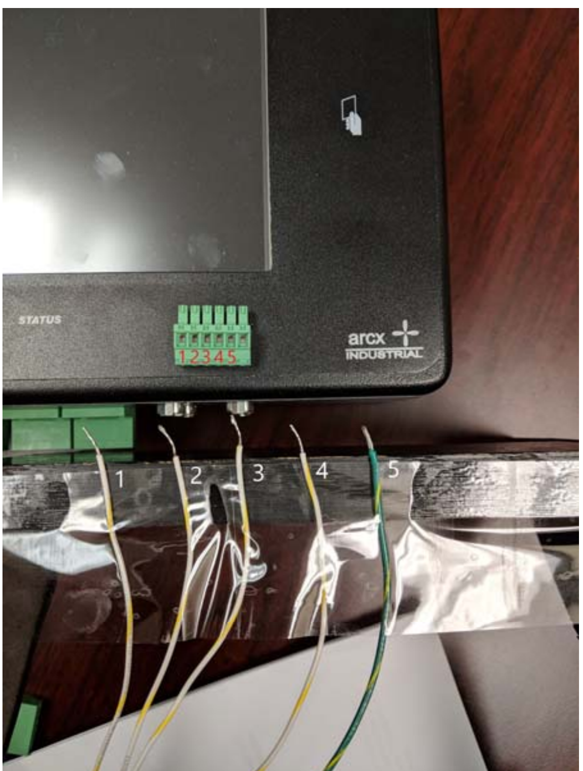


Figure 17

This shows the opposite end of each wire 1 through 5. Screw the wire into the desired port of the terminal block, then plug the terminal blocks into the UP3K where it is labelled inputs as shown in the first figure of this section.

When plugging in the terminal blocks, make sure the left joystick goes to the left inputs (16-13) and the right joystick goes to the right inputs (12-9).

Powering the Box

There are two ways to power the box. One way uses the standard ports, and the other uses power over Ethernet.

Standard Power

Connect the power cable to the UP3K

Figure 18

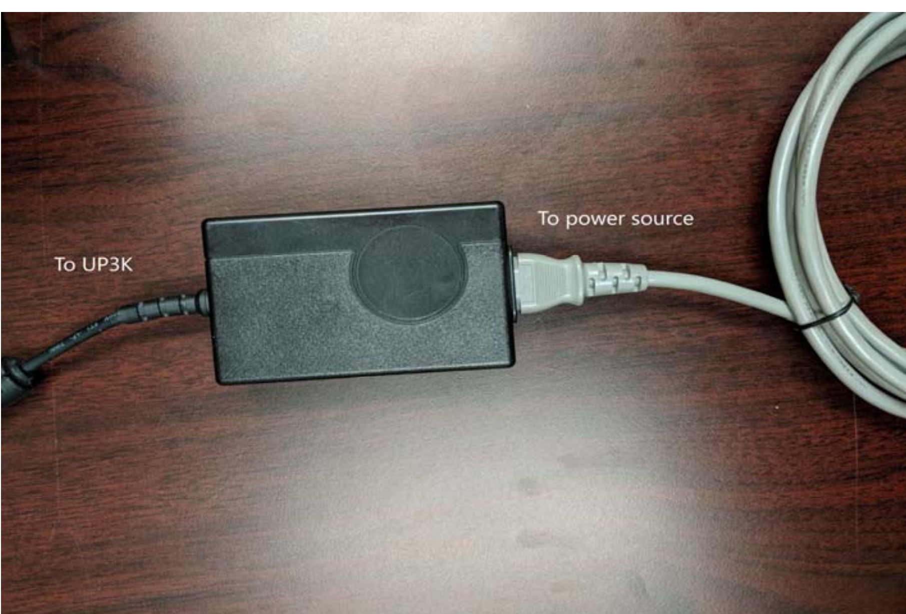


Figure 19

Connect the Ethernet cable to the UP3K and the network source.

Figure 20

Power Over Ethernet

Connect a POE (power over Ethernet) injector to the standard power source and the network source, and then connect the POE line to the Ethernet port on the UP3K.

Figure 21

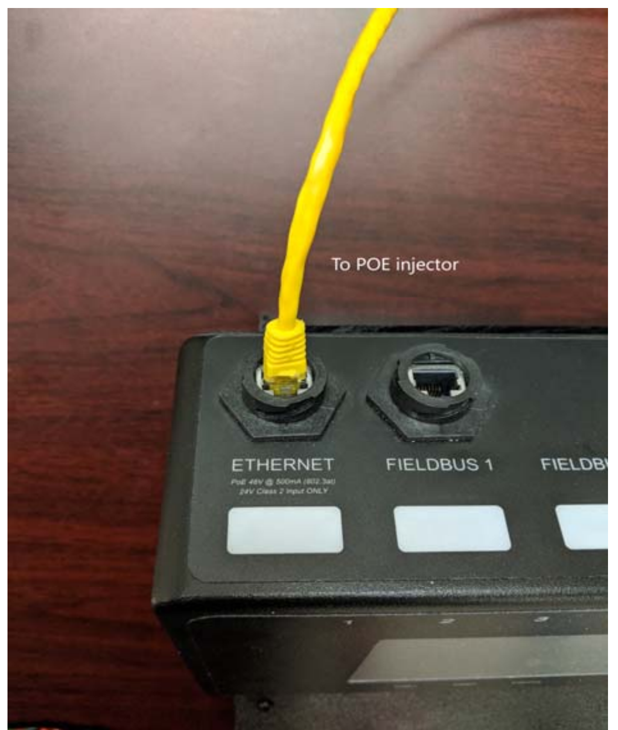


Figure 22

Connecting the box to IIS

Connect the box to the internet, and power it up. The box will turn on.

<Add screen shot>

When this happens, click configuration once the option appears. From this screen, the UP3K's ip address will show, highlighted in red.

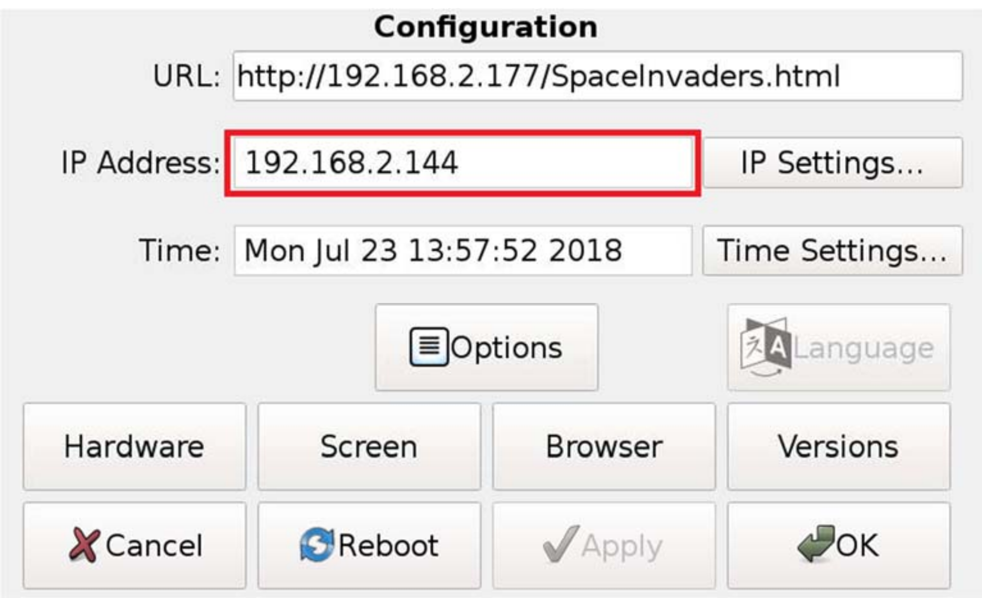


Figure 23

Type the IP into a browser.

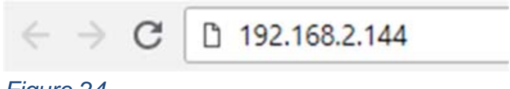


Figure 24

Now the website can be used to access to the box's functions. The code that the box executes can be changed by putting the URL into the "Initial URL" box at the top, highlighted in red.

Figure 25

When the UP3K accesses the initial URL below, it is going to the IP address of the computer, then in the root file, it is looking for the file "SpaceInvaders.html", and it is running the code in that file.

- Make sure the IP address in the 'Initial URL' box is the IP of the machine hosting the server, not the box or a machine that is not related to this interaction.

Code

The rest of this document will explain the code. There will be references throughout that lead to other sections of code. On digital copies, the references can be clicked on to move to the referenced section, and on paper documents, a reference of (001) will be referencing the first section of code which will be titled 001. Names for all variables and functions will be descriptive of their function, so if it is understood what the function does as a whole, continue to understand the current function instead of jumping around the code constantly.

001: HTML and CSS

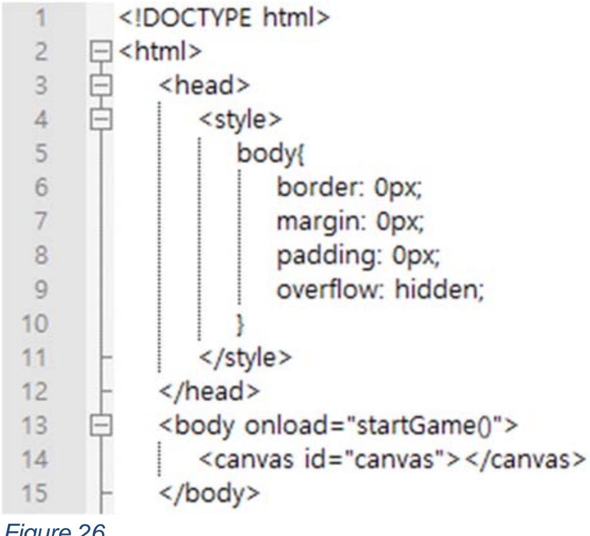


Figure 26

This is where the code starts, from lines 4 to 11, in the style tags, CSS is added, which simply removes the scrollbar and whitespace. Border, margin, and padding being 0 pixels wide ensures that everything made is exactly the expected dimensions. Turning off the overflow means there are no scrollbars if the page takes up the whole screen.

- In line 13, the onload function means that once the whole document has loaded, a function called *startGame* is called. This sets initial conditions for the game and is described in (**Error! Reference source not found.**).

- On line 14, a canvas in html is instantiated, and it is given an id of canvas so it can later be called on in JavaScript.

002: Variable Declaration

```
16  <script>
17      //initializing all the variables used in the game
18      var canvas = document.getElementById("canvas");
19      var ctx = canvas.getContext("2d");
20      var hw = $wat.load('hw')[0];
21      canvas.width = 800;
22      canvas.height = 490;
23      var now;
24      var rand;
25      var player;
26      var pressedButtons = [];
27      var pixelSize = 5;
28      var bulletArray;
29      var bulletHeight = pixelSize*2;
30      var bulletCounter = 0;
31      var bulletSpeed = 6;
32      var lastShot = 0;
33      var playerFireTime = 850;
34      var offset = 60;
35      var alienCount = 0;
36      var alienArray;
37      var alienWidth = pixelSize*11;
38      var alienHeight = pixelSize*8;
39      var alienSpeed = 1.2;
40      var spriteNum;
41      var xAliens = 8;
42      var yAliens = 5;
43      var alienBulletCounter=0;
44      var alienBulletArray;
45      var alienLastShot = 0;
46      var alienFireTime = 1000;
47      var barrierArray=[];
48      var edgeContact = 0;
49      var direction = 1;
50      var level = 1;
51      var score = 0;
```

Figure 27

This section initializes global variables used in the program. This is done to facilitate easier to read code. Variables are named accurately, and reading this will give understanding of some of the variables used later.

There are 5 lines that are currently vital, lines 18 to 22:

- Line 18 creates a JavaScript variable of the html canvas from (**Error! Reference source not found.**) line 14. This can now be called on easily

- Line 19 gets the canvas' context, which allows drawing and renders it in 2D.

- Line 21 and 22 set the width and height of the canvas, which is set to the dimensions of the UP3K screen.

- Line 20 is related directly to the UP3K that the program runs on. It imports the hardware so the different inputs for the controller can be accessed.

003: Movement Variables

```
53  //Movement based variables
54  var movementTimer = null;
55  var spriteAnimation = null;
56  var alienJumpDown = 11;
57
```

Figure 28

These variables, as line 53 suggests, are movement based.

- Lines 54 and 55 set up variables for timers that are defined later in (

007). These will call on certain functions on consistent time intervals when they are defines.

- Line 56 defines how much the aliens move down the screen when they contact the edges in (**Error! Reference source not found.**).

004: Sprite Arrays

```
58  //arrays of all the sprites used, 1 means it will be a white square, 0 means it will be black
59  var playerSprite1 = [
60      [0,0,0,1,0,0,0],
61      [0,0,1,1,1,0,0],
62      [1,1,1,1,1,1,1]
63  ]
64  var playerSprite2 = [
65      [0,0,0,1,0,0,0],
66      [0,0,1,0,1,0,0],
67      [1,0,1,1,0,1,0]
68  ]
69  var bulletSprite = [
70      [1],
71      [1]
72  ]
73  var alien11 = [
74      [0,0,1,0,0,1,0,0,1,0,0],
75      [0,0,0,1,0,0,0,1,0,0,0],
76      [0,0,1,1,1,1,1,1,1,0,0],
77      [0,1,1,0,1,1,1,0,1,1,0],
78      [1,1,1,1,1,1,1,1,1,1,1],
79      [1,0,1,1,1,1,1,1,1,0,1],
80      [1,0,1,0,0,0,0,0,1,0,1],
81      [0,0,0,1,1,0,1,1,0,0,0]
82  ]
83  var alien21 = [
84      [0,0,1,0,0,1,0,0,1,0,0],
85      [0,0,0,1,1,1,1,1,0,0,0],
86      [0,0,1,1,0,1,0,1,1,0,0],
87      [0,0,1,1,0,1,0,1,1,0,0],
88      [0,0,0,1,1,1,1,1,0,0,0],
89      [0,0,1,0,1,1,1,0,1,0,0],
90      [0,0,1,0,0,0,0,0,1,0,0],
91      [0,0,0,1,1,0,1,1,0,0,0]
92  ]
93  var alien31 = [
94      [0,0,0,1,1,1,1,1,0,0,0],
95      [0,0,1,1,1,1,1,1,1,0,0],
96      [0,1,0,0,1,1,1,0,0,1,0],
97      [0,1,0,0,1,1,1,0,0,1,0],
98      [0,1,1,1,1,0,1,1,1,1,0],
99      [0,0,1,1,1,1,1,1,1,0,0],
100     [0,0,0,1,1,0,1,1,0,0,0],
101     [0,0,0,1,1,0,1,1,0,0,0]
102 ]
```

```
103    var alien12 = [
104        [0,1,1,0,0,1,0,0,1,1,0],
105        [0,0,0,1,0,0,0,1,0,0,0],
106        [1,0,1,1,1,1,1,1,1,0,1],
107        [1,1,1,0,1,1,1,0,1,1,1],
108        [1,1,1,1,1,1,1,1,1,1,1],
109        [0,0,1,1,1,1,1,1,1,0,0],
110        [0,1,0,0,0,0,0,0,0,1,0],
111        [0,0,1,1,0,0,0,1,1,0,0]
112    ]
113    var alien22 = [
114        [0,0,0,0,0,1,0,0,0,0,0],
115        [0,1,0,1,1,1,1,1,0,1,0],
116        [0,0,1,1,0,1,0,1,1,0,0],
117        [0,0,1,1,0,1,0,1,1,0,0],
118        [0,0,0,1,1,1,1,1,0,0,0],
119        [0,1,1,0,1,1,1,0,1,1,0],
120        [0,1,0,0,0,0,0,0,0,1,0],
121        [0,0,1,0,0,0,0,0,1,0,0]
122    ]
123    var alien32 = [
124        [0,0,1,1,1,1,1,1,1,0,0],
125        [0,1,1,1,1,1,1,1,1,1,0],
126        [0,1,0,0,1,1,1,0,0,1,0],
127        [0,1,0,0,1,1,1,0,0,1,0],
128        [0,1,1,1,1,0,1,1,1,1,0],
129        [0,0,1,1,1,1,1,1,1,0,0],
130        [0,0,1,1,0,0,0,1,1,0,0],
131        [0,0,0,1,0,0,0,1,0,0,0]
132    ]
133    var deadAlien1 = [
134        [0,0,0,0,0,0,0,0,0,0,0],
135        [0,0,0,1,0,0,0,1,0,0,0],
136        [0,0,0,0,1,0,1,0,0,0,0],
137        [0,0,0,0,1,1,1,0,0,0,0],
138        [0,0,0,1,1,1,1,1,0,0,0],
139        [0,0,0,0,1,0,1,0,0,0,0],
140        [0,0,0,1,0,1,0,1,0,0,0],
141        [0,0,0,0,0,0,0,0,0,0,0]
142    ]
143    var deadAlien2 = [
144        [0,0,0,0,0,1,0,0,0,0,0],
145        [0,1,0,1,1,0,1,1,0,1,0],
146        [0,0,1,1,0,0,0,1,1,0,0],
147        [1,0,0,0,0,0,0,0,0,0,1],
148        [0,0,1,0,0,0,0,0,1,0,0],
149        [0,0,1,1,1,0,1,1,1,0,0],
150        [0,0,0,1,0,0,0,1,0,0,0],
151        [0,0,1,0,0,1,0,0,1,0,0]
152    ]
```

```
153  var barrierSprite1 = [
154      [0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0],
155      [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0],
156      [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0],
157      [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0],
158      [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
159      [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
160      [1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1]
161  ]
162  var barrierSprite2 = [
163      [0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0],
164      [0,0,0,1,1,1,1,1,1,1,1,1,1,0,1,1,0,0,0],
165      [0,1,1,1,1,1,1,1,1,1,1,1,0,0,1,1,1,1,0],
166      [0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0],
167      [1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1],
168      [1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1],
169      [1,1,1,1,0,1,1,1,0,0,0,1,1,1,1,1,1,1,1]
170  ]
171  var barrierSprite3 = [
172      [0,0,0,0,0,1,0,1,1,1,1,1,1,0,0,0,0,0,0],
173      [0,0,0,1,1,1,0,01,1,1,1,1,0,1,1,0,0,0],
174      [0,1,1,1,1,1,1,0,1,1,1,1,0,0,0,1,1,1,0],
175      [0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,0],
176      [1,1,1,1,1,0,1,1,1,0,0,1,1,1,1,0,1,1,1],
177      [1,1,1,1,0,0,1,1,1,0,1,1,1,1,1,1,1,1,1],
178      [1,1,1,1,0,1,1,1,0,0,0,1,1,1,1,1,1,1,1]
179  ]
180  var barrierSprite4 = [
181      [0,0,0,0,0,1,0,1,1,0,0,1,1,0,0,0,0,0,0],
182      [0,0,0,1,1,1,0,01,1,1,1,1,0,1,1,0,0,0],
183      [0,0,1,1,1,1,1,0,1,1,1,1,0,0,1,1,1,1,0],
184      [0,0,0,0,1,1,1,1,1,1,0,1,1,1,0,1,0,0,0],
185      [1,1,1,1,1,0,1,1,1,0,0,1,1,1,1,1,1,1,1],
186      [1,0,1,1,0,0,1,1,0,0,1,1,1,1,1,0,0,1,1],
187      [0,0,0,1,0,1,1,1,0,0,0,1,1,1,0,0,1,0,1]
188  ]
189  var barrierSprite5 = [
190      [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
191      [0,0,0,1,0,1,0,0,0,0,1,0,1,0,1,1,0,0,0],
192      [0,0,1,1,1,1,1,0,1,1,1,1,0,0,1,0,1,0,0],
193      [0,0,0,0,1,1,1,0,1,0,0,1,0,1,0,1,0,0,0],
194      [0,1,1,0,1,0,1,1,1,0,0,1,0,1,1,1,1,0,0],
195      [0,0,1,1,0,0,0,1,0,0,1,1,1,0,1,0,0,1,0],
196      [0,0,0,0,0,1,1,1,0,0,0,1,0,1,0,0,1,0,0]
197  ]
198  var barrierSprite6 = [
199      [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
200      [0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0],
201      [0,0,1,0,0,0,1,0,0,0,0,0,0,1,0,1,0,0],
202      [0,0,0,0,0,1,0,0,1,0,0,1,0,1,0,1,0,0,0],
203      [0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,1,0,1,0],
204      [0,0,1,0,0,0,1,00,0,0,0,0,0,1,0,0,0,0],
205      [0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0]
206  ]
```

Figure 29

This group of code is all sprites, made manually.

```
var barrierSprite1 = [
    [0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0],
    [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0],
    [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0],
    [0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1],
    [1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1]
]
```

Figure 30

This is barrier1, the default state of a barrier, with all of the 1s highlighted. This is an array of arrays, that is, vertically; there are 7 arrays, each containing 19 horizontal values, creating a grid that can be used for a 2d shape. The 1s define the shape of the barrier, and the 0s define where the barrier is not. In *drawSprites* at (**Error! Reference source not found.**), there will be more detail about how it works, but essentially, all the 0s will be a 5x5 pixel square (as determined by pixelSize defined in (**Error! Reference source not found.**)), and all the 1s will be a 5x5 pixel square coloured white.

005: Objects

```
208 //objects, defines the player, bullets, aliens, and barriers
209 function Player(x,y){
210     this.sprite=playerSprite1;
211     this.x=0;
212     this.y=canvas.height-32;
213     this.width=pixelSize*7;
214     this.height=pixelSize*3;
215     this.speed = 4.5;
216 }
217 function Bullet(x,y){
218     this.x=x;
219     this.y=y;
220     this.width=pixelSize;
221     this.height=pixelSize*2;
222 }
223 
224 function Alien(x,y,size,sprite, spriteId){
225     this.x = x;
226     this.y = y;
227     this.size = size;
228     this.sprite = sprite;
229     this.spriteId=spriteId;
230     this.width=pixelSize*7;
231     this.height=pixelSize*5;
232     this.alive=true;
233 }
234 function Barrier(x,y){
235     this.x=x;
236     this.y=y;
237     this.sprite=barrierSprite1;
238     this.spriteId = 1;
239     this.width=pixelSize*19;
240     this.height=pixelSize*7;
241 }
242 
```

Figure 31

This code defines objects for all the elements of the game. The objects contain all of the vital information in one grouping. This makes two steps easier: now there isn't a clutter of variables called playerSpeed, playerSprite, playerXCoordinate, etc, and objects can be called on with player.*attribute*.* This helps keep the code clean and readable.

Another purpose of this is when there are many similar objects with only a few differing properties (like position). The variables x and y defined in all of these objects are the coordinates of the top left corner of the object (the object will later be drawn using a sprite assigned from (**Error! Reference source not found.**)), and using objects like this there can be a list of aliens that all hold the same features but have differing coordinates. These are assigned by defining

them either in the object function or passing them through. The keyword “this” defines an object attribute. More on spriteId and sizes in (**Error! Reference source not found.**).

The player x and y are set to 0, meaning that the player starts at the left side of the screen, and canvas.height-32, starting the player 32 pixels from the bottom

006: Start Game

```
245  //puts the game in a starting position. Canvas is defined and all basic elements are placed
246  function startGame(){
247      alienArray = [];
248      alienCount = 0;
249      bulletArray = [];
250      alienBulletArray = [];
251      barrierArray = [];
252      ctx.fillStyle="#000000";
253      ctx.fillRect(0,0,canvas.width,canvas.height);
254      ctx.fillStyle="#FFFFFF";
255      player = new Player();
256      drawSprite(player.sprite, player.x,player.y,pixelSize);
257      for(var i =0;i<yAliens;i++){
258          for(var k = 0;k<xAliens;k++){
259              if(i==0){
260                  sprite = alien31;
261                  spriteNum = 31;
262              }
263              else if(i==1 || i==2){
264                  sprite = alien21;
265                  spriteNum = 21;
266              }
267              else if(i==3 || i==4 || i==5){
268                  sprite = alien11;
269                  spriteNum = 11;
270              }
271              alienArray[alienCount] = new Alien(k*offset,i*offset, 3,sprite, spriteNum);
272              alienCount++;
273          }
274      }
275      for(i=0;i<4;i++){
276          barrierArray.push(new Barrier(canvas.width*.2*(i+1)-pixelSize*12, canvas.height*0.86));
277          drawSprite(barrierArray[i].sprite,barrierArray[i].x,barrierArray[i].y,pixelSize);
278      }
279      drawAliens();
280      startTimers();
281  }
282
```

Figure 32

startGame sets up all the initial conditions.

- Line 247 and 248: an empty array is initialized that will later be filled with alien objects from (005), but there are no aliens yet.
- 251 and 252 do the same for bullets, and 253 does the same for barriers. These variables are initialized and will be filled farther down.

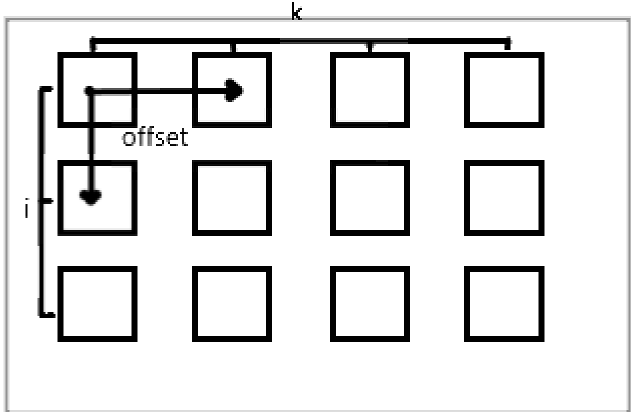
- Line 252 and 253: Fills the entire canvas black, which is in this case, is the entire screen. This makes the background of the game black.
- Lines 254 to 256 make the drawing white, create a player, and then draw the sprite of the player using *drawSprite* (024). This starts the player in the bottom left corner as described at the end of (006).

- Lines 257 to 274 create all the initial aliens. The resulting array of aliens will be a 1D array, but all the aliens require different x and y coordinates.

yAliens was defined in (002) as the number of aliens that appear vertically, and xAliens is defined to be the number of aliens that appear horizontally. There are two nested for loops, i going through yAliens, and k going through xAliens. This is to say, for all i rows of aliens, there are k aliens per row. Depending on the i value, meaning what row it is in, a sprite is assigned, which is why aliens at different heights have different sprites.

With the nature of the array incrementing by 1, all aliens will be 1 pixel away from each other, so the coordinate is multiplied by an offset, which creates a space between all the aliens. Now the aliens can be created. The aliens are assigned the x and y coordinates made with k*offset and i*offset, give it a size of 3 (the overall size of the alien sprite). The alien is assigned a sprite, and assigned a sprite number so the sprite can be changed in (022).

- Line 272 increments to keep track of the number of aliens present. This is needed because the game is won when there are 0 aliens.



*The squares would be replaced with sprites defined in (004)

- Lines 275 to 278 create barriers. For all four barriers, line 276 pushes (adds) a barrier to an array and places it evenly spaced across the screen.

- Line 277 draws the barrier on to the canvas using *drawSprite* (024).

- The function ends by drawing the aliens (017) and starting the timers to run the game (003).

007: Starting and Stopping Timers

```
677  function startTimers(){
678      movementTimer = setInterval(runGame, 3);
679      spriteAnimation = setInterval(changeSprites, 500);
680  }
681  function stopTimers(){
682      clearInterval(movementTimer);
683      movementTimer = undefined;
684      clearInterval(spriteAnimation);
685      spriteanimation = undefined;
686  }
```

Figure 33

This will have two parts, the timers, and the running of the game. First addressed is the timers. The format of a timer is fairly simple:

- Create a variable, and then inside the parentheses, give a function and a time.
- For movement timer, the time is 3 milliseconds, and the function is *runGame*.

This means that every 3 milliseconds, *runGame* will execute. *spriteAnimation* means that every 500 milliseconds, or half second, *changeSprites* (022) will run. *stopTimers* takes these timers, and clears them, and sets them to undefined. The interval is cleared after the player wins or loses so that everything pauses in its place (020).

```
285  //runs all game function
286  function runGame(){
287      resetCanvas();
288      updateDisplay();
289      movePlayer();
290      moveAliens();
291      drawBarriers();
292      moveBullets();
293      checkEndGame();
294  }
```

This leads to *runGame*, which runs every 3 milliseconds when the game is running. This refers to each function in order to put everything in place.

- *Reset canvas*: (008)
- *Update display*: (008)
- *Move player*: (009)
- *Move aliens*: (012)

- *Draw barriers*: (017)
- *Move bullets*: (018)
- *Check endgame*: (019)

008: Background Display

```
296  //clears the canvas and sets it black (ie the background)
297  function resetCanvas(){
298      ctx.clearRect(0,0,canvas.width,canvas.height);
299      ctx.fillStyle="#000000";
300      ctx.fillRect(0,0,canvas.width,canvas.height)
301  }
302
303  //updates all UI items
304  function updateDisplay(){
305      ctx.font = "13px Arial";
306      ctx.textAlign = "left";
307      ctx.fillStyle = "#FFFFFF";
308      ctx.fillText("Level: " + level + "      Score: " + score,0,10);
309  }
310
```

Figure 34

resetCanvas is similar to (006) lines 254 and 255, but with one extra step.

- Line 298 clears the canvas, getting rid of everything on it, leaving it blank white.
- Line 299 sets the colour of our next drawing to be black
- Line 300 fills the entire screen black.

updateDisplay is the small text in the upper left that shows what level the player is on and what the score is. The font is set to size 13 Arial. The text is set to show up on the left, and it is set to be filled in with the colour white. After this, text is written in displaying the level and score 0 pixels from the left side and 10 pixels from the top.

009: Player Movement

```
311  //all interactions involving player input
312  function movePlayer(){
313      findPressedButtons();
314      findUnpressedButtons();
315      if(pressedButtons[0] == 1){
316          shoot();
317      }
318      if(pressedButtons[1] == 1 && player.x >= 0){
319          player.x-=player.speed;
320      }
321      else if(pressedButtons[2] == 1 && player.x + player.width <= canvas.width){
322          player.x+=player.speed;
323      }
324      ctx.fillStyle="#FFFFFF";
325      drawSprite(player.sprite, player.x,player.y,pixelSize);
326  }
```

Figure 35

This function handles all the player's movement

- Lines 313 and 314 will be handled in (010)
    - After these, it is assumed that the joystick is being held actively
    - If it is being held up, the player shoots. More detail on this in (011)
    - If the joystick is being held left, and the player is not yet at the left edge of the screen, the player moves to the left by the player's speed, which was declared in (005)
    - If the joystick is being held to the right and the right side of the player is to the left of the right side of the screen, the player moves to the right by that same speed.

After this the drawing colour is set to white, and in line 325 the player is drawn onto the canvas where its new coordinates are. Keep in mind that at the time of this happening, the canvas has just been set black and drawn the score in the top left only. The last position of the player has been cleared, so only its new position is seen.

010: Finding Pressed Buttons

```
327   //activates buttons on key press
328   function findPressedButtons(){
329       if(hw.digInputs[10].state == true){
330           pressedButtons[0] = 1;
331       }
332       if(hw.digInputs[8].state == true){
333           pressedButtons[1] = 1;
334       }
335       if(hw.digInputs[9].state == true){
336           pressedButtons[2] = 1;
337       }
338   }
339   //deactivates buttons on key unpress
340   function findUnpressedButtons(){
341       if(hw.digInputs[10].state == false){
342           pressedButtons[0] = 0;
343       }
344       if(hw.digInputs[8].state == false){
345           pressedButtons[1] = 0;
346       }
347       if(hw.digInputs[9].state == false){
348           pressedButtons[2] = 0;
349       }
350   }
```

Figure 36

These two functions could be merged into one with the UP3K, but this is more closely related to how it would be done on a keyboard. Since in (002) line 20 the UP3K hardware was imported, digital inputs can be accessed. First check what input the controller is plugged into, and subtract 1, since JavaScript is zero based. From that point check that input's state, and if it is true, then that means the controller is triggering that input. The three inputs the controller is using are checked to see if they are active, and if they are, it sets the index of *pressedButtons* to 1. In *findUnpressedButtons*, inactive buttons are found, and the same indexes are set to 0. This means that later, the array holds all the information of pressed and unpressed buttons, and no further checks are required.

- Index 0 is shooting, index 1 is moving left, and index 2 is moving right.

011: Shoot Function

```
352  function shoot(){
353      now = new Date();
354      now = now.getTime();
355      if(now > lastShot + playerFireTime){
356          bulletArray[bulletCounter%20] = new Bullet(player.x+pixelSize*3,player.y);
357          bulletCounter++;
358          lastShot = now;
359      }
360  }
361
```

Figure 37

As defined in (009) line 316, if the joystick is held in the upward position this function will run. First it gets the current date, and the next line converts that date into time in milliseconds. In (006) line 250, the fire timer for players is set, that is, how long the player has to wait to shoot again. The last shot either never occurred, where it would be 0, or is the time in milliseconds of the previous shot. If must be determined if the current time, in milliseconds, is higher than the time of the last shot in seconds plus the time it takes for the player to fire. If it is not, the program will do nothing.

If it is larger, the player will shoot a bullet. First there is a bullet counter, which is the total number of bullets shot since the start of the level. There is an array of bullet objects, and when the player shoots, a bullet is added to that array at the index of the number of bullets that have been shot, starting back at 0, every 20 shots to save memory. The bullet is created at the position of the horizontal center of the player, and at the player's height. After this, the time of the previous shot to the time of this shot, to make sure another shot doesn't happen until the fire time is up.

012: Alien Movement

```
362  //handles alien movement
363  function moveAliens(){
364      checkForHits();
365      alienShoot();
366      //bounces the aliens off the edges
367      for(i=0;i<alienArray.length;i++){
368          if(alienArray[i].x+alienWidth-25 >= canvas.width && alienArray[i].alive==true){
369              direction = -1;
370              edgeContact=1;
371          }
372          else if(alienArray[i].x <= 0 && alienArray[i].alive==true){
373              direction = 1;
374              edgeContact=1;
375          }
376      }
377      alienSpeed = direction*Math.abs(level/3+0.6+2/alienCount)*3;
378      //moves the aliens down if they have contact with an edge
379      for(i=0;i<alienArray.length;i++){
380          alienArray[i].x+=alienSpeed;
381          if(edgeContact==1 && alienArray[i].alive==true){
382              alienArray[i].y+=alienJumpDown;
383          }
384      }
385      //alien speed at certain intervals and draws them onto the canvas
386      edgeContact=0;
387      drawAliens();
388  }
```

Figure 38

The function *moveAliens* covers all movement covered with aliens, and all object interaction. Line 364 and 365 will be covered in (013) and (016), but essentially *checkForHits* handles all objects coming in contact with each other. This includes bullets, barriers, and aliens. *alienShoot* is the function that covers the aliens shooting back at the player.

Line 367 to 376 handle contact with the edges. The for loop on 367 checks all the aliens, and the if statement on 368 says that if an alien hits the right side of the screen, and the alien is alive (not undergoing its death animation), the direction is made negative and edgeContact states that there has been contact with an edge. The other statement on 372 says that if an alien has hit the left side of the screen and is alive (not undergoing its death animation), the direction it travels is made positive and edgeContact states that there has been contact with an edge. After this the speed is determined by the following equation:

$$direction * Math.abs(\frac{level}{3} + 0.6 + \frac{2}{alienCount}) * 3;$$

- The direction, based on the last wall hit, is multiplied by the absolute value so that no other variables can ever have an effect on this.

- Level / 3 makes the speed get somewhat faster every level.

- 0.6 is a base speed that is added to make the game always fun, not dead slow on level 1

- And 2/alienCount means that as aliens die, the remaining aliens will speed up, starting small, and getting very fast at the end.

The code from Lines 379 to 384 moves the aliens. The for-loop scrolls through all the aliens and moves them horizontally by the defined speed. After this, if one of the previous statements says that there is contact with the edge, the aliens move down by a distance defined in (002).

Now edgeContact is reset to 0. This means that it will be 0 until another alien next hits an edge, and means that aliens don’t keep moving down forever once they touch an edge once. Now the aliens are drawn in their position, further discussed in (017).

013: Hit Registration

```
389    function checkForHits(){
390        //resolves all contact with aliens
391        if(bulletArray.length > 0){
392            for(i=0;i<alienArray.length;i++){
393                for(j=0;j<bulletArray.length;j++){
394                    if(collides(alienArray[i],bulletArray[j]) && alienArray[i].alive==true){
395                        addScore(alienArray[i]);
396                        alienArray[i].sprite=deadAlien1;
397                        alienArray[i].spriteId=1;
398                        alienArray[i].alive=false;
399                        bulletArray.splice(j,1);
400                        alienCount--;
401                        break;
402                    }
403                }
404            }
405        }
406        //resolves all contact with barriers
407        for(i=0;i<barrierArray.length;i++){
408            for(j=0;j<alienArray.length;j++){
409                if(collides(barrierArray[i],alienArray[j])){
410                    barrierArray.splice(i,1);
411                    break;
412                }
413            }
414            for(j=0;j<bulletArray.length;j++){
415                if(collides(barrierArray[i],bulletArray[j])){
416                    barrierArray[i] = decayBarrier(barrierArray[i]);
417                    bulletArray.splice(j,1);
418                    break;
419                }
420            }
421            for(j=0;j<alienBulletArray.length;j++){
422                if(barrierArray[i]!=null&&collides(barrierArray[i],alienBulletArray[j])){
423                    barrierArray[i] = decayBarrier(barrierArray[i]);
424                    alienBulletArray.splice(j,1);
425                    break;
426                }
427            }
428        }
429    }
```

Figure 39

Line 390 makes sure there are bullets, because if there aren't, looping through the array of bullets would throw an error.

- Lines 392 and 393: for each alien, check all of the bullets
- Line 394, if they collide (023), the function handles how the objects are supposed to interact.

First the score for the aliens is added, which will be further described in (014). The alien's sprite is set to the start of its death animation, and its spriteId is set to 1. After this, alive is set to false, meaning the bullet can't collide with the alien again, and a dead alien wont trigger all the aliens to switch directions. Then splice the bullet array, which removes the bullet from that position, so the bullet stops existing on contact with an alien. After, the number of aliens in alienCount is decreased, then the loop is exited.

Next, check everything that involves barriers, which have lots of interaction. For every barrier, each alien is checked, making the barrier disappear if it contacts an alien. After this, each bullet is checked. If it contacts the barrier, the barrier breaks a bit (015), and the bullet disappears. Next check each alien bullet (016), and it interacts with the barrier in the same way a player bullet does.

014: Score Handling

```
430  function addScore(deadAlien){
431      var levelAdd = level*5;
432      if(deadAlien.spriteId == 31 || deadAlien.spriteId == 32){
433          score+=25+levelAdd;
434      }
435      else if(deadAlien.spriteId == 21 || deadAlien.spriteId == 22){
436          score+=15+levelAdd;
437      }
438      else if(deadAlien.spriteId == 11 || deadAlien.spriteId == 12){
439          score+=5+levelAdd;
440      }
441  }
```

Figure 40

Since a simple comparison between sprites is not possible, a spriteId is assigned to track what sprite each object has using an integer.

This function is started by knowing an alien is dead and which alien it is. The program takes this alien, checks its spriteId, and based on that, adds a certain number to the score. Additionally, as a reward to players for making it to higher levels where the aliens are harder to kill, additional points are added based on the level the player is at.

Aliens with spriteId of 31 or 32 are the top level, and will earn most points. Aliens with a spriteId of 11 or 12 are the lowest aliens, and will give the lowest score.

015: Barrier Decay

```
442  //decays barriers based on life
443  function decayBarrier(barrier){
444      switch(barrier.spriteId){
445          case 1:
446              barrier.sprite = barrierSprite2;
447              barrier.spriteId=2;
448              break;
449          case 2:
450              barrier.sprite = barrierSprite3;
451              barrier.spriteId=3;
452              break;
453          case 3:
454              barrier.sprite = barrierSprite4;
455              barrier.spriteId=4;
456              break;
457          case 4:
458              barrier.sprite = barrierSprite5;
459              barrier.spriteId=5;
460              break;
461      }
462      return barrier;
463  }
```

Figure 41

If a bullet hits a barrier, the barrier will be passed into the function *decayBarrier*. The object's spriteId will be determined and increased by 1. The sprite is then set to the next sprite, defined in (004), and the barrier will appear more eroded.

016: Alien Shooting

```
464    //makes aliens shoot every second
465    function alienShoot(){
466        rand = Math.floor(Math.random() * (alienArray.length));
467        now = new Date();
468        now = now.getTime();
469        if(now > alienLastShot+ alienFireTime){
470            alienLastShot = now;
471            for(i=0;i<alienArray.length;i++){
472                if(i==rand){
473                    alienBulletArray[alienBulletCounter%20] = new Bullet(alienArray[i].x+alienArray[i].width,alienArray[i].y);
474                    alienBulletCounter++;
475                }
476            }
477        }
478    }
```

Figure 42

This function is very similar to shoot at (011). The only difference is that instead of defaulting the bullet to the player's position, it picks a random number between 0 and the number of aliens, then makes the alien at that index shoot a bullet. The value is stored in alienBulletarray instead of the regular bulletArray. The difference comes in *moveBullets* in (018).

017: Drawing Aliens and Barriers

```
479  //draws an array of aliens based on their position in a grid
480  function drawAliens(){
481      ctx.fillStyle="#FFFFFF";
482      if(alienArray.length>0){
483          for(var i = 0;i<alienArray.length;i++){
484              if(alienArray[i]!=null){
485                  drawSprite(alienArray[i].sprite,alienArray[i].x, alienArray[i].y,alienArray[i].size);
486              }
487          }
488      }
489  }
490
491  function drawBarriers(){
492      if(barrierArray.length>0){
493          for(i=0;i<barrierArray.length;i++){
494              if(barrierArray[i]!=null){
495                  drawSprite(barrierArray[i].sprite,barrierArray[i].x,barrierArray[i].y,pixelSize);
496              }
497          }
498      }
499  }
500
```

Figure 43

Functions *drawAliens* and *drawBarriers* are near identical, spare the fact that one deals with an array of aliens, and the other an array of barriers. They check if there are any aliens or barriers, because if there aren't, there is no point in executing the rest. They check all the aliens and barriers, and if the position in each array isn't empty, it will draw the sprite (024) of the object in its location.

018: Bullet Movement

```
501  //moves player and alien bullets
502  function moveBullets(){
503      //player bullets
504      if(bulletArray.length>0){
505          for(i=0; i<bulletCounter;i++){
506              if(bulletArray[i]!=null){
507                  bulletArray[i].y-=bulletSpeed;
508                  ctx.fillStyle="#FFFFFF";
509                  drawSprite(bulletSprite,bulletArray[i].x,bulletArray[i].y,pixelSize,pixelSize);
510              }
511          }
512      }
513      //alien bullets
514      if(alienBulletArray.length>0){
515          for(i=0; i<alienBulletArray.length;i++){
516              if(alienBulletArray[i]!=null){
517                  alienBulletArray[i].y+=bulletSpeed*0.7;
518                  ctx.fillStyle="#FFFFFF";
519                  drawSprite(bulletSprite,alienBulletArray[i].x,alienBulletArray[i].y,pixelSize,pixelSize);
520              }
521          }
522      }
523  }
```

Figure 44

Function *moveBullets* deals with bullets shot by the player and from aliens. First, the function addresses bullets shot by the player, so, all indexes of bulletArray are checked. For all that contain a bullet, the bullet is moved up by a speed assigned in (002), and its sprite is drawn. (024)

After this the same process occurs for alien bullets. All indexes of alienbulletArray are checked, and for all that contain a bullet, the bullet moves down by 70% of the player's bullet speed, and their sprite is drawn (024).

019: Endgame Conditions

```
528  //checks for conditions of victory and loss
529  function checkEndGame(){
530      //if all aliens are dead
531      if(alienCount == 0){
532          winGame();
533      }
534      //if the player is dead
535      for(i=0;i<alienArray.length;i++){
536          if(collides(player, alienArray[i]) || alienArray[i].y+alienArray[i].height > canvas.height){
537              loseGame();
538              break;
539          }
540      }
541      for(i=0;i<alienBulletArray.length;i++){
542          if(alienBulletArray[i] != null && collides(alienBulletArray[i],player)){
543              loseGame();
544              break;
545          }
546      }
547  }
```

Figure 45

Function *checkEndGame* tests the conditions for win or loss. The condition for winning the level is that all the aliens are dead, so if the count of aliens is 0, the player has won, and can start the next level(020).

There are three ways for the player to die: the aliens hit the player, the player gets hit by a bullet, and the aliens reach the bottom of the screen.
Line 535 tells if the player has been hit by an alien. It checks if any alien hits the player (023), and if this happens, it runs the *loseGame* function (020).

In the same way, it checks if any of the alien's bullets hit the player, and if this happens, it runs the *losegame* function (020).

020: Winning and Losing

```
548  //continue on to the next round showing the round and score if the player kills all the aliens
549  function winGame(){
550      clearInterval(movementTimer);
551      clearInterval(spriteAnimation);
552      score+=200;
553      playerFireTime-=85;
554      alienFireTime-=75;
555      level++;
556      ctx.font = "50px Arial";
557      ctx.textAlign = "center";
558      ctx.fillStyle = "#FFD700";
559      ctx.fillText("Level: " + level, canvas.width/2, canvas.height/3);
560      ctx.fillStyle = "#000000";
561      ctx.strokeText("Level: " + level, canvas.width/2, canvas.height/3);
562      ctx.fillStyle = "#FFD700";
563      ctx.fillText("SCORE: " + score, canvas.width/2, canvas.height/2);
564      ctx.fillStyle = "#000000";
565      ctx.strokeText("SCORE: " + score, canvas.width/2, canvas.height/2);
566      setTimeout(startGame, 2500);
567  }
568  //lose game screen if the player dies
569  function loseGame(){
570      stopTimers();
571      player.sprite=playerSprite2;
572      resetCanvas();
573      updateDisplay();
574      drawSprite(player.sprite, player.x,player.y,pixelSize);
575      drawAliens();
576      drawBarriers();
577      ctx.font = "50px Arial";
578      ctx.textAlign = "center";
579      ctx.fillStyle = "#FFD700";
580      ctx.fillText("SCORE: " + score, canvas.width/2, canvas.height/2);
581      ctx.fillStyle = "#000000";
582      ctx.strokeText("SCORE: " + score, canvas.width/2, canvas.height/2);
583      setTimeout(offerRestart, 2500);
584  }
```

Figure 46

First *winGame* will be discussed.
Initially the timers are stopped. Next some basic variable handling occurs. Leveling up will:

- Add 200 to the score
- Make the player fire faster
- Make the aliens fire slightly faster

- Increase the level
    - Aliens also have their movement speed in part based on level (012)

Next printing text is handled, this has been described in (008), but the *winGame* and *loseGame* sizing is slightly different. First, all the text is set to 50 pixels large.

- In line 557, text is aligned to the center. Because of this alignment, wherever text is placed, it will be centered and not taking up differing space depending on the text width.

- Lines 558 and 559 print the level ⅓ of the way down the screen in gold.

- Lines 560 and 561 outline this in black.

- Lines 562 to 565 do the same thing but for the score, and perfectly in the middle of the screen.

- Line 566, runs the function *startGame* after 2500 milliseconds, or 2.5 seconds.

loseGame runs in a very similar fashion. This one stops for an indefinite amount of time, so *stopTimers* is called (003) to avoid background running. The player's sprite changes to one of a broken player ship (004) to show that the player has lost. After this, it resets the canvas (008), prints the score and level in the top left (008), and redraws the player without moving them (024). After this the function draws the aliens and barriers (017) so that the player can see exactly what hit them and how they lost. Now the score is printed in the center of the screen the same as in *winGame*, so the player can see how well they did. After 2500 milliseconds, or 2.5 seconds, the player is offered an option to restart the game (021).

021: Restart Conditions

```
580  function offerRestart(){
581      ctx.font = "20px Arial";
582      ctx.textAlign = "center";
583      ctx.fillStyle = "#FFD700";
584      ctx.fillText("HOLD RIGHT AND SHOOT FOR RESTART", canvas.width/2, canvas.height*3/4);
585      ctx.fillStyle = "#000000";
586      ctx.strokeText("HOLD RIGHT AND SHOOT FOR RESTART", canvas.width/2, canvas.height*3/4);
587      movementTimer = setInterval(restartGame, 350);
588  }
589  function restartGame(){
590      findPressedButtons();
591      findUnpressedButtons();
592      if(pressedButtons[0] == 1 && pressedButtons[2] == 1){
593          playerFireTime = 850;
594          alienFireTime = 1000;
595          stopTimers();
596          level = 1;
597          score = 0;
598          startGame();
599      }
600  }
```

Figure 47

The first function *offerRestart* happens 2.5 seconds after the player loses, and it creates text in the same way as in (020), giving the player instructions to restart the game. Now, every 350 milliseconds the game will run *restartGame*.

The function first checks for all the buttons the user is and is not holding down (010). If the player is holding the joystick both up and to the right, the fire speed resets, the timers are stopped (003), and the level is set back to 1, and the score to 0. Then, *startGame* (006) is run, which will run everything at its initial conditions, completely restarting the game. If the player is not holding the buttons, nothing happens.

022: Sprite Animation

```
606   //handles all sprite animation in place. Alien change and death
607   function changeSprites(){
608       for(i=0;i<alienArray.length;i++){
609           switch(alienArray[i].spriteId){
610               case 31:
611                   alienArray[i].sprite = alien32;
612                   alienArray[i].spriteId = 32;
613                   break;
614               case 32:
615                   alienArray[i].sprite = alien31;
616                   alienArray[i].spriteId=31;
617                   break;
618               case 21:
619                   alienArray[i].sprite = alien22;
620                   alienArray[i].spriteId=22;
621                   break;
622               case 22:
623                   alienArray[i].sprite = alien21;
624                   alienArray[i].spriteId=21;
625                   break;
626               case 11:
627                   alienArray[i].sprite = alien12;
628                   alienArray[i].spriteId=12;
629                   break;
630               case 12:
631                   alienArray[i].sprite = alien11;
632                   alienArray[i].spriteId=11;
633                   break;
634               case 1:
635                   alienArray[i].sprite=deadAlien2;
636                   alienArray[i].spriteId=0;
637                   break;
638               case 0:
639                   alienArray.splice(i,1);
640                   break;
641           }
642       }
643       for(i=0;i<barrierArray.length;i++){
644           switch(barrierArray[i].spriteId){
645               case 5:
646                   barrierArray[i].sprite = barrierSprite6;
647                   barrierArray[i].spriteId=6;
648                   break;
649               case 6:
650                   barrierArray.splice(i,1);
651                   break;
652           }
653       }
654   }
```

Figure 48

In (003), timers were set, and one of them was for changing sprites. This function runs every 500 milliseconds, or every half second. Every alien is checked in a for loop, and the alien’s spriteId is checked so the program knows what sprite they are. Based on this sprite, it is switched to another sprite, which makes it look animated.

If an alien dies, its spriteId is changed to 1. This is the first step of a death animation. This will change to 0, the second part of the death animation, and it will then disappear and be spliced out of the array (013).

Barriers are only active here if they reach case 5. They don't animate until their death animation, where they will hit their last 2 sprites, then be spliced out similarly to the aliens.

Note: sprites are shown in (004).

023: Collision Function

```
655     //helper function for collision to help code be neat
656     function collides(object1, object2){
657         if(object2==null)
658             return false;
659         else if(object2.x+object2.width>object1.x
660             && object2.x<object1.x+object1.width
661             && object2.y<object1.y+object1.height
662             && object2.y+object2.height>object1.y){
663                 return true;
664         }
665         else{
666             return false;
667         }
668     }
```

Figure 49

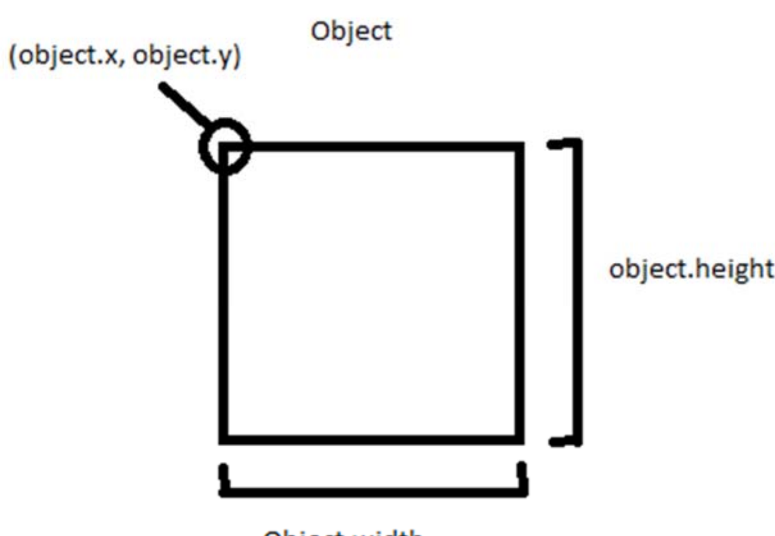


Figure 50

This is the general structure for an object, it has an x, based on its distance from the side of the screen, and it has a y, based on its distance from the top of the screen. These are the different parts of the object:

- Object.x is the left side of the object
- Object.x+object.width is the right side of the object

- Object.y is the top of the object
- Object.y+object.height bottom of the object

In line 656 the function gets two objects, and names them object 1 and object 2 for the generalization of the function. First it checks if object 2 is null, which is checked so the function does not return an error for no object existing. If it is null, it cannot collide, false is returned (no collision).

In line 659 the function starts a multi-line if statement. The four bullet points above are further explained in a more understandable language.

Note: (0,0) is the top left corner of the screen. X increases to the right, and Y increases downwards.

Else if(

- The right side of the second object is farther right than the left side of the first object
- The left side of the second object is farther left than the right side of the first object
- The top of the second object is above the bottom of the first object
- The bottom of the second object is lower than the top of the first object

)

If all of these conditions are true, then one object's hitbox is inside another's hitbox, so there is a collision. If this is the case, the function will return true (collision).

If all of these conditions are not true, then it means one object's hitbox is not inside another's hitbox, so there is no collision. If this is the case, the function will return false (no collision).

024: Sprite Drawing

```
669     //draws the defined sprites where they should be
670     function drawSprite(sprite, posx, posy, size){
671         for(var height=0;height<sprite.length;height++){
672             var spriteX = sprite[height];
673             for(var width=0;width<spriteX.length;width++){
674                 if(sprite[height][width] == 1){
675                     ctx.fillRect((width*size)+posx,(height*size)+posy,size+1,size+1);
676                 }
677             }
678         }
679     }
```

Figure 51

Function *drawSprite* takes in four variables. The first is the sprite assigned to the object; it will be from the list in (004). After it will take posx, which is the object's x coordinate, and posy, which is the object's y coordinate. Next size is inputted, which is the size desired for each pixel.

This is a nested for loop creating a grid, similar to in (006). The function creates this grid using white squares where a 1 is found in the array, and empty squares where there is no 1. Recall in (004) where there was a barrier sprite with all the 1's highlighted, showing the shape of the barrier.

The function goes through each row, creating an array for that row, calling it spriteX. Searching through this array, where there is a 1, a white rectangle is printed. Once each row is searched, the sprite will be completely printed.

The white rectangle is drawn using:

fillRect(x coordinate of top left corner, y coordinate of top left corner, x length, y length);

where:

- X coordinate: width * size puts the white rectangle in its place in the sprite horizontally. Adding posx puts the sprite in its place on the entire game screen
- Y coordinate: height * size puts the white rectangle in its place in the sprite vertically. Adding posy puts the sprite in its place on the entire game screen
- (for both) size + 1 is the length of the edge of the square. Add 1 to this pixel width so the player can't see the grid of computer pixels underneath the sprite model, and it looks solid.